Final Report

Rent Payment Made Simple

Developing software in an entrepreneurial environment

Version 1.1 **November 28, 2011**

Delft University of Technology
Faculty EEMCS

Bachelors Project Group (IN3405)
Dirk Guijt 1308084
Lars Tijhuis 1308335

Thesis Committee

Chairman:	Prof. Drs. Dr. L.J.M. Rothkrantz
B.Sc. Coordinator:	Drs. P.R. van Nieuwenhuizen
Company Supervisors:	R. Nguyen, S. Kanajan

Summary

Roomplug was founded to make renting simple. It provides all the tools needed to go from having an empty property to getting the lease agreement signed. Roomplug also wants to solve the problems on rent payment. That is what this project is all about: creating an online rent payment system.

Since Roomplug is a startup company, it is expected that the requirements will change as the business model and needs of Roomplug change over time. The Feature Driven Development (FDD) methodology was chosen to create a development track suitable for this changing set of requirements. In the FDD track, the features are split up and grouped in milestones. Every next milestone can be changed to support the (new) needs of Roomplug.

Before work on the first milestone starts, a global high-level design is created to make sure that all milestones work towards the same goal. The design details of each milestone are then created as soon as work on that milestone starts. This prevents unnecessary design work on features that will never be implemented.

Implementation starts after the milestones design is ready. The Roomplug back-end system, and thus the payment system, is written in PHP using Zend Framework and its MVC pattern. It uses a MySQL database to store all the data.

Testing the code after the implementation of a milestone is a big part of the project. Many test classes are written to ensure the correct functioning of the payment system. PHPUnit generates extensive coverage reports after each test run. All this is very use full to ensure that code from earlier milestones still works after making changes and additions to implement the next milestone.

The finished prototype of the payment system lets users create single custom payments and set up recurring payments using PayPal as a payment method. Although an account is needed to receive payments, paying your rent can also be done as a guest without an account. The system is designed in such a way that other payment systems could easily be added in the future.

The payment system is currently running live on the Roomplug beta site. It is ready to be used by the rental market.

Preface

This report is the final report concluding the development of an online rent payment system for Roomplug LLC located in San Clemente, California, USA. This project was executed for course *IN3405 - Bachelorproject*, a part of the bachelors curriculum of Computer Science at Delft University of Technology (DUT) in The Netherlands. The project spanned 12 weeks starting at July 18, 2011 and ending on October 10, 2011. The first two orienting weeks were carried out in Delft, The Netherlands, after which project execution moved to San Clemente. This report will give insight in how the project's development track was carried out while being abroad in an entrepreneurial environment.

We would like to thank the following people for their contribution and support throughout the project:

- Rodney Nguyen (Roomplug), for being our direct supervisor during development of the payment system, for helping us get to know the Roomplug systems, and for showing us various parts of the American culture;
- Sri Kanajan (Roomplug), for being our supervisor, for providing us with a place to call home, and for taking us to various places to meet up with the American people;
- Leon Rothkrantz (DUT), for being our supervising professor in The Netherlands, and for the support throughout the project;
- Peter van Nieuwenhuizen (DUT), for coordinating the bachelors project, and helping us to get everything set up on the DUT side of the project;
- Jan de Vries (DUT), for helping us arranging everything to do this project in the United States, and for helping us out applying for the grants;
- Eric Bouwers (SIG), for providing us with useful feedback on the maintainability of our code.

Dirk Guijt and Lars Tijhuis

San Clemente, California, USA, October 2011.

Table of contents

Summary ... 3
Preface ... 5
Table of contents ... 7
1. Introduction ... 9
  1.1 Roomplug ... 9
  1.2 Problem definition ... 9
  1.3 Development methodology ... 10
  1.4 Schedule ... 11
  1.5 Report structure ... 12
2. Technologies ... 13
  2.1 Zend Framework ... 13
    2.1.1 Zend_Db ... 13
    2.1.2 Zend_Layout ... 13
    2.1.3 MVC structure ... 14
    2.1.4 Evaluation ... 14
  2.2 PayPal API ... 15
    2.2.1 Adaptive Payments ... 15
    2.2.2 Pre-Approved Payments ... 16
    2.2.3 Verify account status ... 17
    2.2.4 Instant Payment Notification ... 17
  2.3 PHPUnit ... 18
3. Requirements ... 19
  3.1 Requirements development ... 19
  3.2 Functional requirements ... 19
  3.3 Quality requirements ... 20
  3.4 Platform requirements ... 20
  3.5 Process requirements ... 20
  3.6 Features and milestones ... 20
    3.6.1 Milestone 1: Basic payments (core) ... 20
    3.6.2 Milestone 2: Recurring payments (additional) ... 21
    3.6.3 Milestone 3: Make the payment system an independent tool ... 22
4. Design ... 25
  4.1 Global design ... 25
    4.1.1 Zend Framework and modelling ... 25
    4.1.2 The global model ... 26
  4.2 Milestone 1 ... 29
    4.2.1 Model ... 29
    4.2.2 Controller and View ... 30
    4.2.3 MVC ... 32
  4.3 Milestone 2 ... 33

4.3.1 Changes in the model ... 33
4.3.2 Changes in the controllers and views ... 34
4.4 Milestone 3 ... 34
4.5 The existing Roomplug system ... 35
4.6 User Interface ... 35
5. Implementation ... 41
5.1 Design workaround Zend Framework ... 41
5.2 PayPal API ... 43
5.2.1 Pay request ... 43
5.2.2 *Preapproval* request ... 45
5.2.3 getVerifiedStatus request ... 45
5.3 Software Improvement Group feedback ... 46
5.3.1 Examples; Unit interfacing ... 47
5.3.2 Examples; Unit size ... 48
5.3.3 Feedback on improvements ... 49
5.4 Maintainability ... 49
6. Testing ... 51
6.1 Test plan ... 51
6.1.1 Automated testing ... 51
6.1.2 Regression testing ... 51
6.2 PHPUnit ... 52
6.3 Results ... 54
6.3.1 Pros vs Cons ... 54
6.3.1.1 Finding bugs ... 55
6.3.1.2 Refactoring ... 56
6.3.1.3 Dependency on PayPal ... 56
6.3.2 Coverage and complexity reports ... 56
6.4 Future test automation ... 57
7. Conclusion ... 59
7.1 Results ... 59
7.2 Recommendations ... 61
7.2.1 Features ... 61
7.2.2 Future work ... 62
References ... 65
Definitions and Abbreviations ... 67
Appendices ... 69

Introduction

This chapter will describe what problem is solved in this project, how that problem is solved and in what time span this is done. Chapter 7 will revisit this chapter to draw conclusions on the results of this project.

1.1 Roomplug

Roomplug is a startup company founded in 2010 with the mission of making online property management accessible and easy for everyone [1]. The online rental market already had property marketplaces, background and credit check services and rent payment systems available. Roomplug is unique because it puts all these tools and services together into one system. From finding a room/roommate to signing the lease agreement and paying the rent. Everything is there, it's a one-stop shop for renting.

Currently the Roomplug system provides the following functionality:

1. Users can create advertisements for the rooms they have for rent, or to indicate that they are looking for a room;
2. Users can search trough the advertisements;
3. Users can apply to an advertisement, after which the owner of the advertisement can accept or decline an application or ask them for a background report;
4. Users can use the TransUnion [2] background check service to provide the landlords with a background report about themselves;
5. Users can set up a lease agreement with another user. This lease agreement can be digitally signed.

1.2 Problem definition

The project is part of the one-stop shop for renting that Roomplug wants to create. The goal is to make rent payment easy by creating an online payment tool, accessible to everyone. The central question for the one-stop shop is "How can Roomplug take away the pains in the current rental business by providing an online solution for it?". The goals of this project is to contribute to this central question.

The specific problem that needs to be solved by this project is "How can the current payment process between renters and landlords be improved by using the tools and services of the Roomplug website?". The answer to this problem should meet the following high-level requirements:

1. Roomplug wants a prototype of a rent payment system, which allows renters and landlords to pay each other online.

2. The payment system should be designed and build to be a stand alone system, using data from other components of the Roomplug website.

3. Roomplug users should be able to receive payments from everyone, even if the paying person does not have a Roomplug account.

4. Roomplug users should be able to set up recurring payments to pay their rent automatically.

5. Roomplug users should be able to manage their payments. This managing could include, for example, an overview of the payment history.

6. The online rent payment system should be designed to support multiple payment methods. PayPal is the method that is first to be adopted by the online payment system.

Although there are many solutions to transfer money digitally, Roomplug chose to only include PayPal as a payment method for now. Setting up the connection to the PayPal APIs and processing the transactions being executed in the PayPal systems, is part of this project.

1.3 Development methodology

Working in an entrepreneurial environment makes things different compared to doing a project at a big established company. While big corporate businesses have solid strategies worked out for the next few years, startup companies are struggling with their business plan as a whole. As Roomplug is getting closer to the market and as they talk to more and more future investors, things change around. The business model and the priority of feature requirements change constantly. This makes it harder for a software engineer to plan out a 2 month development track, as is the case in this project. The used development methodology and the design choices should take these possible changes into account.

Because Roomplug is a startup company, the Feature Driven Development (FDD) is the most suitable method for the task at hand. Following the FDD method, the required system is split up into various different features. This makes it possible to switch feature priority around later on in the development track, as the requirements and feature necessity changes. FDD is an Agile form of developing [3].

First, out of the requirements, a feature list is made, stating what features should be implemented to comply the requirement. Those features can be grouped in related sets and subject areas. During this project, each set of features is considered a milestone, working from milestone to milestone, starting with the most fundamental ones. After the features and milestones are defined, an overall design is made with a high-level object model and notes as result. For each feature set, a more detailed design is made prior to implementation. Also, a set of tests will be written to test the features. This can either be done after implementation, or parallel to the implementation process.

After all features are implemented, the project is completed. Failure to finish all features, will leave the system uncompleted, but it will still be able to work. This also illustrates the importance for the order in which features are implemented since there is only limited time in this project.

1.4 Schedule

The FDD methodology ensures that the final product will still work even if a certain feature isn't implemented. However, this project is aiming to finish most of the features in the given time. Since the project is being executed on location in the United States, there is little margin in the time constraints. To work as efficient as possible and get as much work done as possible, a carefully scheduled development track is required. The schedule that is created can be found in Figure 1.1. A textual description of the schedule is provided bellow the schedule.

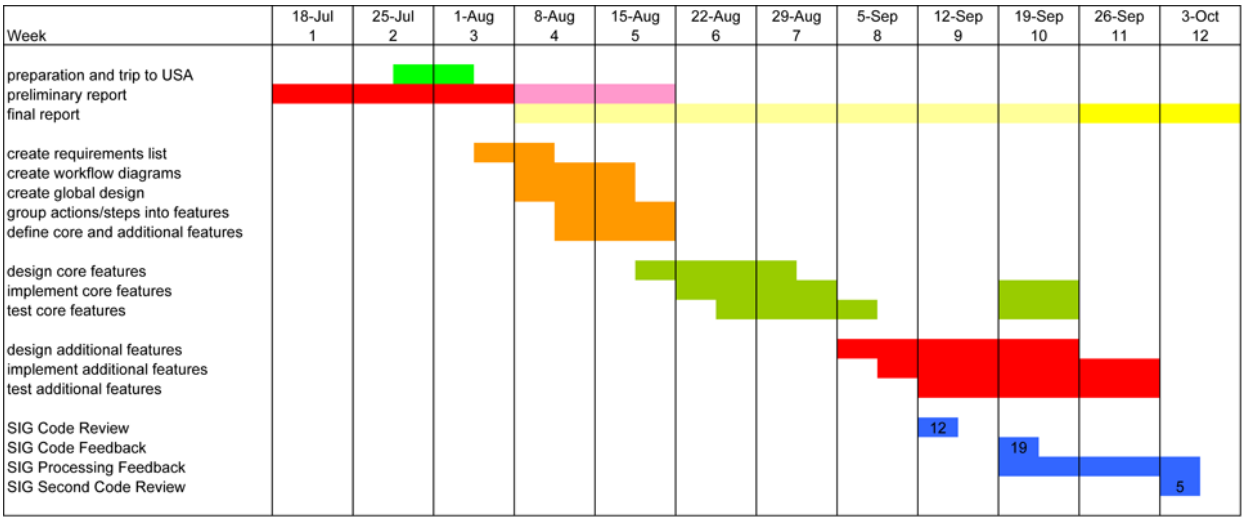

Figure 1.1: The time schedule for the development of the payment system

Weeks 1-3:

- Get to know the development environment;
- Research existing peer-to-peer payment systems;
- Review PayPal APIs;
- **Deliverables**: Preliminary report.

Weeks 4-5:

- Make overall design, high-level object model and feature list;
- Group features in sets;
- Determine the order of implementation;
- Determine actual milestones and deadlines for them.

Weeks 6-8:

- Make feature level design and implementation for core features;
- Write tests for core features;
- **Deliverables**: Code for review by the Software Improvement Group.

Weeks 8-12:

- Make feature level design and implementation for extending features;
- Write tests for the extending features;
- **Deliverables**: Final report.

1.5 Report structure

This report is structured in the same way as the development track for this project is structured. Chapter 2 gives insight into the technologies that are used to solve the problem. A basic understanding of these technologies is needed to do any work on the project. Chapter 3 discusses the requirements of the project and it shows how the problem is divided into features and milestones. Chapter 4 shows how the design of the project was constructed. Chapter 5 will give some implementation details and it shows how the comments of the Software Improvement Group were used to improve the code. Chapter 6 discusses the testing of the payment systems classes. Chapter 7 will finally draw conclusion on the result of the project and give recommendations on future work.

Technologies

This chapter will give insight in the technologies that are used in this project. This chapter discusses parts of Zend Framework, the PayPal API and PHPUnit.

2.1 Zend Framework

The system is built using Zend Framework. This is a popular PHP framework that allows the programmer to use a lot of components like a database access layer, a template engine and much more. One of its bigger advantage is the modular MVC design to make code more reusable and easier to maintain [4]. It was an obvious choice to use Zend Framework, since the current system is built in it and it would be tedious to deviate from this.

Zend_Db, *Zend_Layout* and the MVC structure were used extensively while building the payment system. Those parts will be discussed into more detail in the next sections. Also, an evaluation is written in section 2.1.4 about how the Zend Framework contributed to the project.

2.1.1 Zend_Db

Zend_Db is a library that contains all components needed for communication with the database [5]. It starts of with the *Zend_Db_Adapter* component. This component provides the connection to the database. One of the big advantages of using this adapter class, is the type of database can easily be replaced by another [6]. The current system is already set up for communicating with a MySQL database.

Classes of importance in this project are the *Zend_Db_Table* and *Zend_Db_Table_Row* classes. Those classes provide a way to communicate with data from the database. The *Zend_Db_Table* class represents a table, allowing easy retrieval of data from the table. For the data retrieval either queries can be written, or a query can be build using the *Zend_Db_Select* object. The data retrieved by the *Zend_Db_Table* class is returned in the form of a *Zend_Db_Table_Row* object. Using this object, it is easy to edit the data and write the changes to the database by calling a simple *save* function. [7, 8, 9]

In this project, the abstract versions of those classes are used to implement more specific getters and setters and additional functions. More details on how the *Zend_Db* classes were used in the design can be found in section 4.1. Implementation details can be found in section 5.1.

2.1.2 Zend_Layout

Zend_Layout provides a way to create template files in which you define a page layout and its placeholders. Using this allows a programmer to only set up the layout once and then define the contents for each page in different files [10].

The current system does not utilize this tool. However, since the payment system uses the same general layout and the target is to make it more maintainable, the layout is stripped from the original source files and placed in a layout file. Placeholders are added and therefore the payment system view files only need to take care of the content.

2.1.3 MVC structure

The MVC structure is one of the biggest advantages of Zend Framework. It allows the programmer to easily separate presentation, business logic and data access [11]. The directories and file structure are organized in such a way as to keep the three parts separate.

The basic flow in the MVC structure goes as follows. The *Zend_Controller_Front* intercepts an incoming request and calls the corresponding action in the specific *Zend_Controller* that implemented the action. This controller then retrieves the needed data from the model, using the *Zend_Db* classes described in section 2.1.1. The data to be presented is set in the parameters of a *Zend_View* object, which contains the logic to display this data. The *Zend_View* scripts typically contain HTML with placeholders in it for the variable data to display [12, 13].

2.1.4 Evaluation

The use of the Zend Framework had some advantages, but also meant some work had to be done that could have been achieved using other tools.

The MVC structure provided by the Zend Framework is very powerful. It provides an easy solution by keeping the different parts of the MVC structure in separate files. This part of the Zend Framework alone, makes it worth using it.

A weaker point is the Zend_Db library. Although the classes provide an easy connection with the database and its tables, numerous tools are available mapping database tables to objects with easy to use functions. Those tools take away the need to write the different getters and setters for each of the properties in a table. Although Zend Framework allows you to use tools like these on top of the framework, they were not used in this project since this would imply changing the use of the database throughout the whole (existing) system.

Zend_Layout makes the use of templates easy; it does what it should do. Of course, there are several other template engines with more extensive features, but there was no need for any more advanced features in this project.

As mentioned before, the MVC structure provided by the Zend Framework alone, makes it worth using it. For new projects, additional tools replacing some of the features might be taken into consideration.

2.2 PayPal API

PayPal provides developers with extensive APIs to access their services. The services used in this project, are those to make payments, set up pre-approvals, verify a users PayPal account and confirming processed payments. The first two can be accessed using the Adaptive Payments API. Verifying a users PayPal account is done with the Adaptive Accounts API and confirming the processing of payments is done using a service called Instant Payment Notification. Usage of those APIs require approval from PayPal. The payment process was therefore analyzed twice by PayPal and both times it got approved. [14]

PayPal also provides SDKs for various programming languages, including PHP. This allows the programmer to use objects to set up and make calls to the API, rather then formatting the call itself. This SDK is also used in this project.

2.2.1 Adaptive Payments

The Adaptive Payment API can be used for various types of payment set ups. Since Roomplug wants to take a small fee when two users send money to each other, the *Chained Payments* option is the most suitable solution. This solution allows us to define a primary receiver and multiple secondary receivers. This way, Roomplug can take their fee from the amount payed, while the paying user only sees the primary receiver, being the other user, as the receiver. The primary receiver will be notified about receiving the money and will then automatically send the fee to Roomplug. This structure is shown in Figure 2.1.

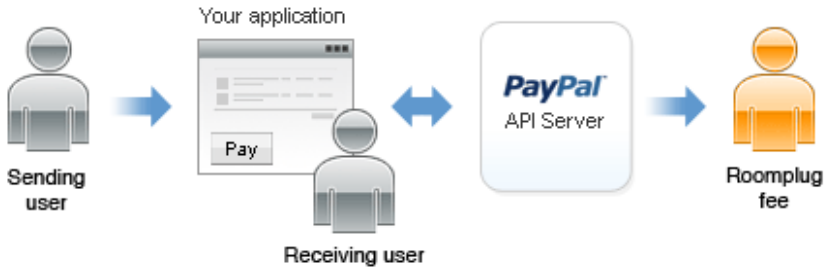

Figure 2.1: The abstract structure for a Chained Payment [14]

Using the SDK, a payment can be made by providing the email and amount to receive of both the primary and secondary receiver. The flow of a payment, as illustrated in Figure 2.2, goes like this;

1. First, the Pay request is constructed, setting the details of the payment to be done. This request is then send to PayPal;
2. PayPal processes the request and returns a token unique to the requested payment;
3. The user is redirected to the PayPal systems where they login and process the payment;
4. After completion (or cancellation), the user is redirected back to the URL provided in the pay request.

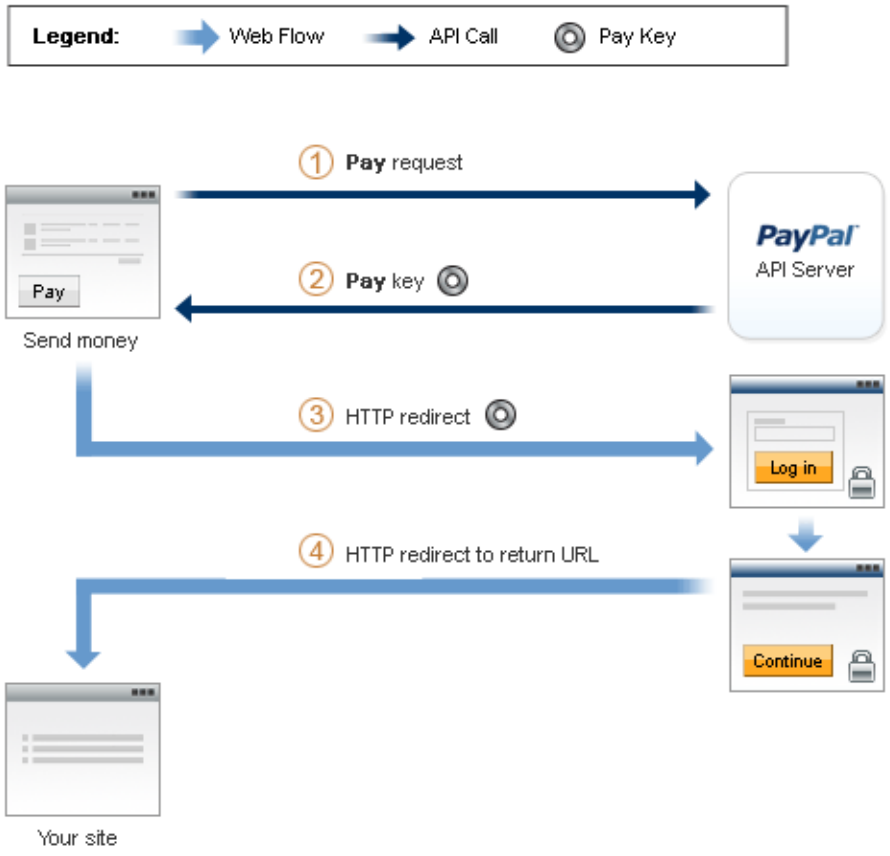

Figure 2.2: The flow for executing a payment through PayPal [14]

More details on implementation of the pay request can be found in section 5.2.1.

2.2.2 Pre-Approved Payments

The pre-approval process is similar to the Pay flow described in section 2.2.1. There are some substantial differences in setting up the request though. Instead of defining a receiver, all you provide is information about the amount of money. The period, the number of occurrences within this period, the amount per payment and the total amount are properties that need to be set for a pre-approval.

Basically, the user gives Roomplug permission to charge the user's PayPal account for the defined amount. After retrieving confirmation of the pre-approval, the token can be used to make payments that follow the information set in the request. This payment process goes like the process illustrated in Figure 2.3. As can be seen in this figure, no user action is required. The Pay request is set up in the same way as a regular payment, except for that an extra property, the pre-approval key, is now also set. More details on implementation of pre-approval requests can be found in section 5.2.2.

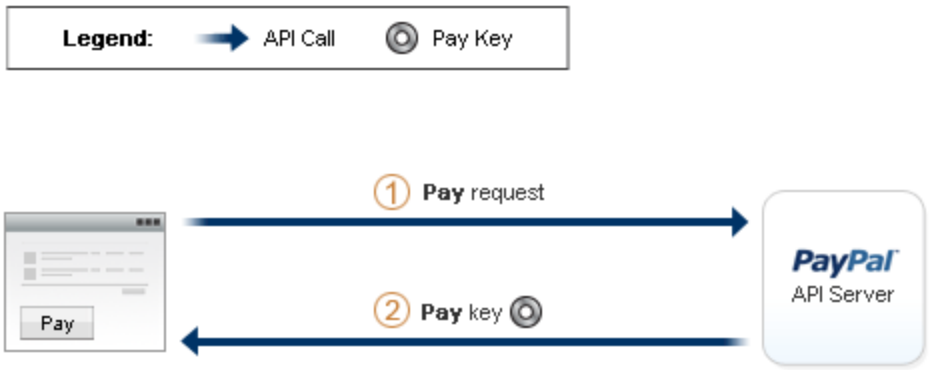

Figure 2.3: Executing a pre-approved payment [14]

2.2.3 Verify account status

A user needs to have a PayPal account in order to receive payments. Once they have a PayPal account, all the information needed is the email address they registered with on PayPal, to send them the money.

Since a user might have registered on Roomplug using a different email address, users can provide their PayPal information through a separate form on Roomplug. Both when providing this information and when initiating a payment, the system needs to make sure the receiving account is available on PayPal. Therefor, the Adaptive Account API is used. Using this API, the status of an account can be verified.

The *GetVerifiedAccount* request requires the programmer to provide the email address and the first and last name of the account to verify. The first and last name have to match the ones provided in the PayPal account of the user. The request simply returns whether the account was verified or not. More on implementation on this request can be found in section 5.2.3.

2.2.4 Instant Payment Notification

For the Roomplug systems to know what is going on at the PayPal side of each transaction, PayPal provides the Instant Payment Notification (IPN) service. For each transaction running through the Roomplug application on PayPal, a notification message is sent to the Roomplug IPN listener. This listener then updates the Roomplug sy[1]stems with the new information. The IPN listener is just a web page running on the Roomplug website on an open URL, which is accessible from the web. To ensure that the system is not subjected to abuse by sending fake calls, the IPN calls need to go through a handshake process shown in Figure 2.4.

[1] A transaction does not necessarily mean that money is being transferred.

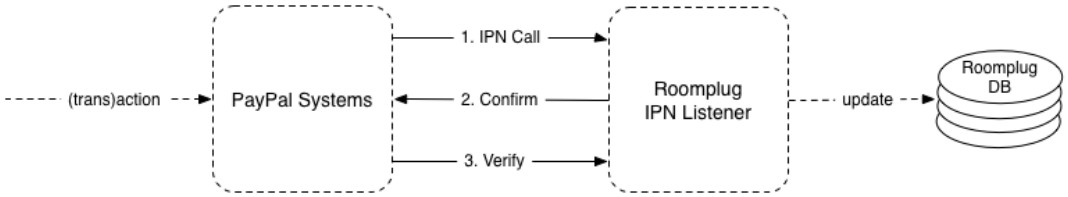

Figure 2.4: The IPN flow

As soon as a transaction is completed in the PayPal systems, they will send Roomplug an IPN call. This call contains information about the transaction itself. For Roomplug, only the payment, pre-approval and pre-approval cancellation transactions[1] are relevant. Once the IPN call reaches Roomplug, the listener wants to confirm that the message was sent by PayPal. Therefore, it makes a call back to PayPal, notifying the PayPal system that it wants to verify the message. PayPal will respond if the call was valid or not. If it is valid, the Roomplug IPN listener can be sure that the message came from PayPal and it can continue with updating the Roomplug systems with the new information.

2.3 PHPUnit

PHPUnit is the unit testing solution for PHP. Its goals are to make tests easy to write, read and execute. It offers mechanisms, such as the *setUp* and *tearDown* functions, to make tests run independent from the current state of the system [15]. PHPUnit also integrates the *PHP_CodeCoverage* library, giving both feedback on the coverage of your code and its complexity [16].

Zend Framework allows the programmer to extend the set of assertions with specific Zend Framework assertions, by using the *Zend_Test_PHPUnit* component. Assertions to check certain information shown on the web page, *assertQuery*, and the controller and action used, *assertController* and *assertAction*, are a good example of those [17].

Although much effort was put into setting up PHPUnit to work with the current system, in the end it helped a lot in finding bugs, fixing them, and to maintain the system while developing new features. More on the specific usage of PHPUnit can be found in section 6.2.

Requirements

In this chapter, the process followed to formalize the requirements, and the outcome of this process, the requirements, are presented. Next to that, the features to meet a requirement are defined and those features are grouped into feature sets. Those sets represent a milestone, resulting in a prioritized list of features to implement.

3.1 Requirements development

Prior to the development of the requirements, extensive research on the rental market and the available online tools was done. The main focus of this research was on the payment process and the information that is required by those systems. The user base was represented by Roomplug itself. The founders already had experience using rental tools, since they are landlords themselves.

After the research, functionality specifications for the payment system were developed. These specifications were discussed with Roomplug and then transformed into requirements. In the end, the requirements were prioritized and divided into development milestones.

3.2 Functional requirements

1. A user should be able to make custom payments to another user;
2. A user should be able to pay to a Roomplug account, both as a guest or as a logged in user;
3. A user should be able to see all of the payments made as a guest once a Roomplug account is created if the user uses the same email address as provided in the guest payment system;
4. A user should be able to link their PayPal account to their Roomplug account;
5. A user must have a Roomplug account with a PayPal account linked to it, in order to receive payments;
6. A user should be able to set up a recurring payment to another user;
7. A user has to have a Roomplug account, in order to set up a recurring payment;
8. A user should be able to choose from at least four different payment frequencies while setting up a recurring payment: weekly, biweekly, first day of the month and last day of the month;
9. A user should be able to view all of his incoming and outgoing payments;
10. A user should be able to view the details of a payment;
11. A user should be able to delete open payments (and open payments only);
12. A user should at least be able to use PayPal as a payment method.

3.3 Quality requirements

13. The payment system may never mark a payment as ‘paid’ if no actual transaction occurred;
14. The payment system may never allow a payment to be paid again if it is marked as ‘paid’ already;
15. The payment system needs to store key information on a transaction so that a payment can be verified manually. That way, a payment can be verified if the system of the payment provider (e.g. PayPal) fails to report a successfully processed payment to the Roomplug system;
16. The payment system must be designed in such a way that new payment methods can be added easily in the future;
17. The payment system should be fully integrated within the current Roomplug system so the user experiences being on the same web application while managing his/her payments. This also means that it is implemented in Zend Framework using the MVC pattern.

3.4 Platform requirements

18. The payment system should run on a CentOS 5.4 linux environment with kernel version 2.6.18-028stab070.14. The environment should run the Apache 2.2.19 web server with PHP 5.2.17 and MySQL 5.0.92.

3.5 Process requirements

19. The payment system should be developed using the feature driven development method;
20. The delivery date for the payment system will be October 10, 2011.

3.6 Features and milestones

Following the FDD method, features are defined for each of the requirements. Those features will be sorted on priority, defining a set of core features and additional features. Also, milestones are defined, creating a milestone by milestone path for implementation. Note that the quality, platform and process requirements are not grouped into specific features, as they need to be taken into account during the whole development process.

The outcome of this grouping and prioritizing is shown in the following sections.

3.6.1 Milestone 1: Basic payments (core)

This milestone (partly) covers requirements 1, 2, 9, 10, 11 and 12, and includes the following features:

Manage payments

Goal: View history, create new and delete existing payments.

Features: View payments overview;
Shows open payments and the payment history.
View payment details;
Shows the specific details of a payment.
Create custom payments;
Create a custom payment by choosing a receiver, the amount and giving a description.
Remove custom payment;
Remove a custom payment; this is only possible when the actual payment has not been completed yet.

Pay a payment

Goal: Allow users to pay created payments through PayPal. The system should be designed in such a way that other payment methods can be easily added.

Features: Pay an open payment;
Do the actual paying of a open payment. This process involves communication with PayPal and going through their work flow.

3.6.2 Milestone 2: Recurring payments (additional)

This milestone covers requirements 6, 7 and 8, but also involves extension of the functionality made for requirements 9, 10, 11 and 12, as those actions should also be available for these kind of payments. The following features need to be implemented:

Set up a recurring payment

Goal: Users should be able to set up recurring payments. This way, the execution of payments can be automated, which is common with rent payment.

Features: Create a recurring payment;
Set up a recurring payment by choosing a receiver, the amount, and giving a description. Next to that, define the frequency for the payments to occur and the start and end date for them to occur.

Activate a recurring payment

Goal: A user needs to authorize Roomplug to make payments on their behalf. After setting up a recurring payment, it needs to be activated by approving Roomplug to make payments through their PayPal account.

Features: Activate a recurring payment;
A created recurring payment needs to be activated. This can be done by setting up a pre-approval on PayPal, allowing Roomplug to charge a fixed price from the users account in a given period.

Create and execute an occurrence of a recurring payment

Goal: The Roomplug system must create occurrences of the recurring payments when needed and execute them.

Features: create and execute occurrences;
On a timely bases, the Roomplug system must check if an occurrence of the active recurring payments is needed. This should be done without any user interaction since the recurring payments are pre-approved.

3.6.3 Milestone 3: Make the payment system an independent tool

Although not stated in the requirements above, the original requirements stated the system should allow users with a lease agreement on Roomplug to be able to make payments to each other. In the process of creating the online payment tool, the importance of this tool to Roomplug became more clear. Thus, the original requirements were changed to allow all users to make payments to each other. This was accomplished in milestone 3, covering and complementing requirements 2, 3, 4 and 5. The features that were implemented in milestone 3 are:

Selecting the receiver

Goal: Instead of selecting the receiver from a dropdown containing the users you have a lease agreement with, the user should now be able to search for a user and select it.

Features: Find user by name;
When looking for a user by name, update the search results on every key stroke using AJAX. This gives a dynamic experience, making it easier to find the right receiver. The information shown should respect the users privacy settings.
Find user by email;
Again, when searching by email, update the search results on every key stroke. Since the user can prefer to keep their email address private, only show search results if the email address looked for is identical to the email address in the database.

Guest payments

Goal: Since the payment system is a profitable tool for Roomplug, it is important to make it as easy as possible to use it. A user not having an actual account should not stop it from making a payment.

Features: Make a payment as a guest;
When going to the payment page while not logged in, show extra fields for the users first and last name and their email address. They can choose to fill in this information or login with their existing account. The rest of the process should be the same as the normal payment process.

Set up your PayPal account

Goal: In former milestones, the email users registered with on Roomplug was used to send the money to on PayPal. This could lead to trying to send money to non existing PayPal accounts. The goal of this feature is to provide users with a form to set up their PayPal account on Roomplug.

Features: Validate account;

To set up your PayPal account on Roomplug, fill in the first and last name and email address used on PayPal. Using the PayPal API this information can be verified and saved when it is valid.

Design

This chapter will discuss both the global design of the payment system and the design by milestone. This gives insight in how every feature was designed while still keeping the bigger picture in mind.

Because this project follows the FDD methodology, a global design is needed to prevent design and implementation problems in the incremental development of features. The more detailed design issues will be solved when arriving at a certain milestone, because these (might) change over time. Especially in a startup environment.

4.1 Global design

The model of the payment system will be implemented in Zend Framework. Using this framework presents a design problem. This problem and the influence on model design in general will be explained first and then the design of the payment system itself is discussed.

4.1.1 Zend Framework and modelling

The information in the objects of the model have to be stored in a database. Zend provides the *Zend_Db* library, which makes it easy to map the PHP model classes on database tables. The developer does not have to worry about connecting to a database and querying, although the developer is still able to do this if needed. The *Zend_Db* library takes care of inserting, updating, deleting and retrieving data from the database.

Unfortunately, the *Zend_Db* library also has its limitations, which influences the construction of the global model. Especially, mapping multiple database tables on one class is an issue. It can be done using a reference map, which is implemented in one table class and references to the properties of the other table(s). While this is not a problem when having only two classes, a single class extending another class for example, it becomes an issue if there are multiple classes that need to reference a single table. The reference map then has to be implemented in all classes. This problem and the solution are illustrated in Figure 4.1.

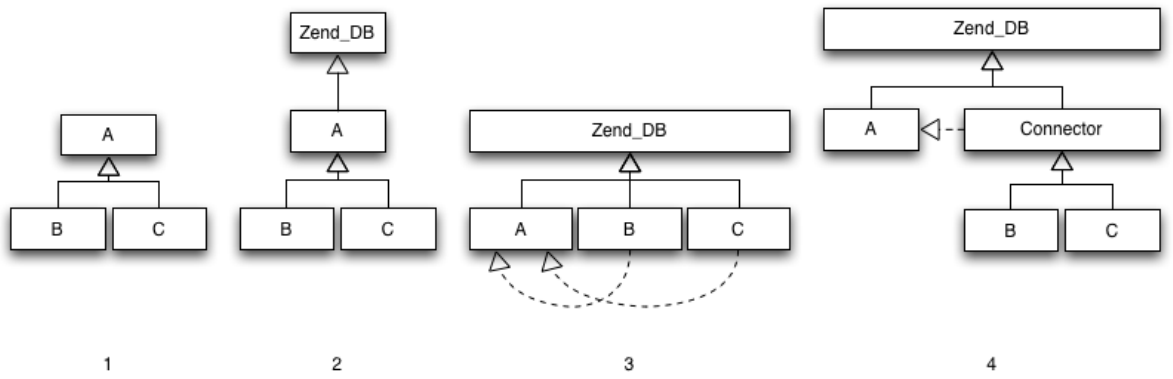

Figure 4.1: Inheritance and the database mapping problem (and solution)

The figure shows four situations. Situation 1 shows the model as it is designed. There are classes B and C, which are extending class A. All three classes have their own database table storing the information of the objects of that type. It would be easy to implement this model and let class A extend the *Zend_Db_Table* class of the *Zend_Db* library, like shown in situation 2. Unfortunately, Zend Framework does not support this.

Letting all the subclasses have a reference map to the super class is a possible solution for this problem. The design for this solution is shown in situation 3. Although that solution works, the reference map has to be implemented in every subclass (both B and C). This extra work has to be done every time a new sub class is added. It is even worse when there are changes in superclass A. In this case, all the subclasses have to update their reference maps.

The solution is to have a connector between the superclass and its subclasses. In this example, the connector can hold the reference map to A. Having just a reference map presents new implementation problems, because the class now has the fields of both tables, but does not have the functions of both classes. Section 5.1 will present a solution to this implementation problem using the model design of situation 4.

4.1.2 The global model

As stated before, the model is relatively simple. There are just payments and recurring payments. These classes both need to support at least PayPal as a payment method but there could be more methods in the future. Hence, the subclasses. The *Payment* class and *RecurringPayment* class will have some properties in common. For example, they both got have paying and a receiving user. So a *BasePayment* is created, holding that information. Including all of this, the basic model will look like Figure 4.2.

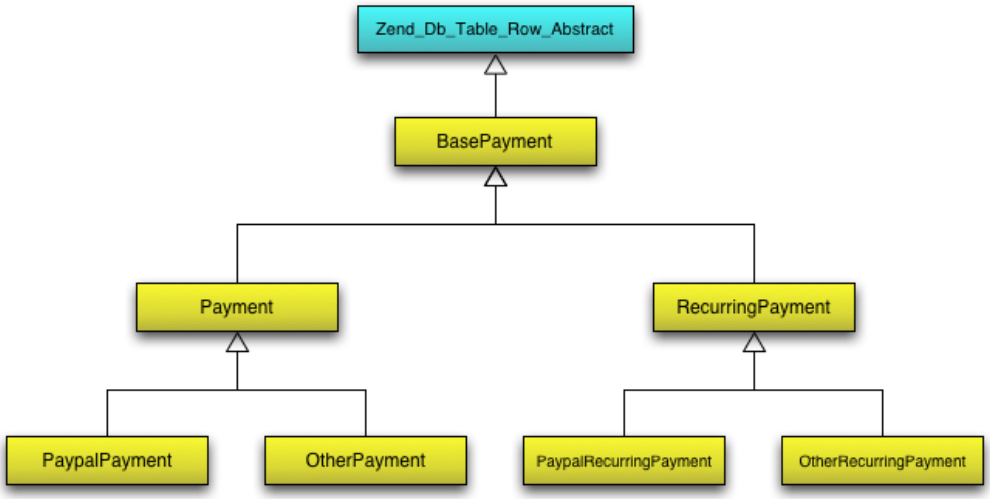

Figure 4.2: The basic model of the payment objects

The model has to be changed to take Zend Framework implementation problems, discussed in section 4.1.1, into account. The classes cannot just be mapped on the database using the structure of Figure 4.2. We have to introduce connectors for every super class. After adding these, the model will look like Figure 4.3.

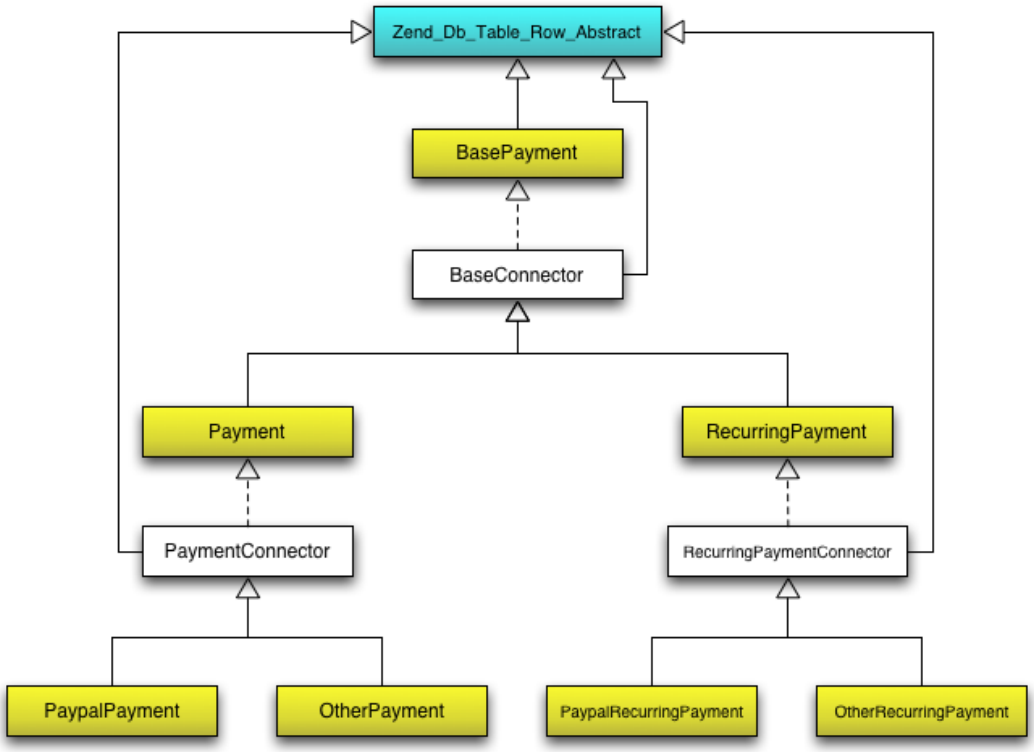

Figure 4.3: The basic model of the payment objects using Zend Framework

The model shown in Figure 4.3 looks a lot more complex than the one shown in Figure 4.2. This complexity makes implementation, testing and maintaining the system a lot harder. To prevent future implementation and maintenance problems, simplifying the model will be necessary. This requires a more closer look at the model properties and inheritance, since the inheritance problem solved by the connectors adds most of the complexity.

Since it is unknown what properties and functions are necessary in order to make payments using other payment systems than PayPal, the model on the *Payment* and *RecurringPayment* subclasses remain untouched.

Then there is the *BasePayment* class. Having knowledge about the properties that the *Payment* and *RecurringPayment* classes share is needed in order to evaluate the necessity of this superclass. In order to build a better global design, more detailed information on these objects is needed. It turns out that there are four properties shared between these classes. They will both have a paying user, a receiving user, the amount that needs to be payed, and a description. Because it is only four properties, the decision is made to leave out the *BasePayment* class and have these properties modelled in both classes. This will benefit the whole development track by simplifying the model.

Now, there is one problem left. There is no link between a recurring payment and the payments that are occurrences of that recurring payment. To solve this, a *RecurringPaymentLink* table and class is introduced to hold information on payments that are occurrences of a recurring payment.

The final model is now complete and is shown in Figure 4.4 and the relationship between the object is shown in Figure 4.5.

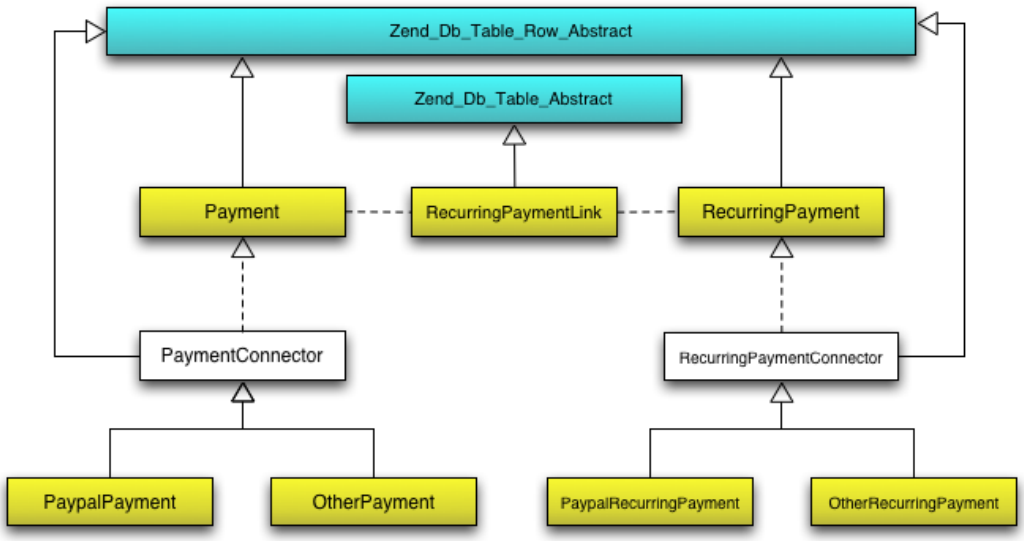

Figure 4.4: The payment system model

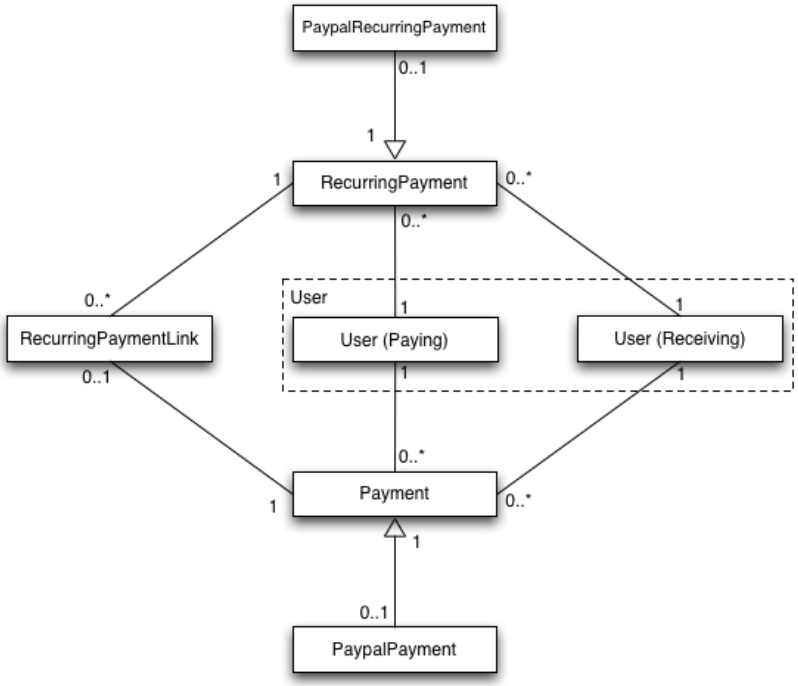

Figure 4.5: The relational model of the payment system

4.2 Milestone 1

The first milestone is about the creation of custom payments. Users must be able to manage their payments and they must be able to pay them trough PayPal. The latter requires communication with PayPal, thus an adapter is created to communicate with PayPal using its SDK. This will be done in this milestone as well. This section will first discuss the model component of the MVC pattern. The payment systems view script only produce HTML, JavaScript and JSON text which map directly on the controllers actions. Therefore, the controllers and the views are modelled as a whole. Every page in the views part of MVC corresponds to an action in the controller part.

4.2.1 Model

Since the global model is already available, the modelling in this milestone is all about specifying the properties and functions. The result of this can be found in the class diagram of the first milestone, which can be found in Appendix G.2. A simplified version of this diagram is shown in Figure 4.6.

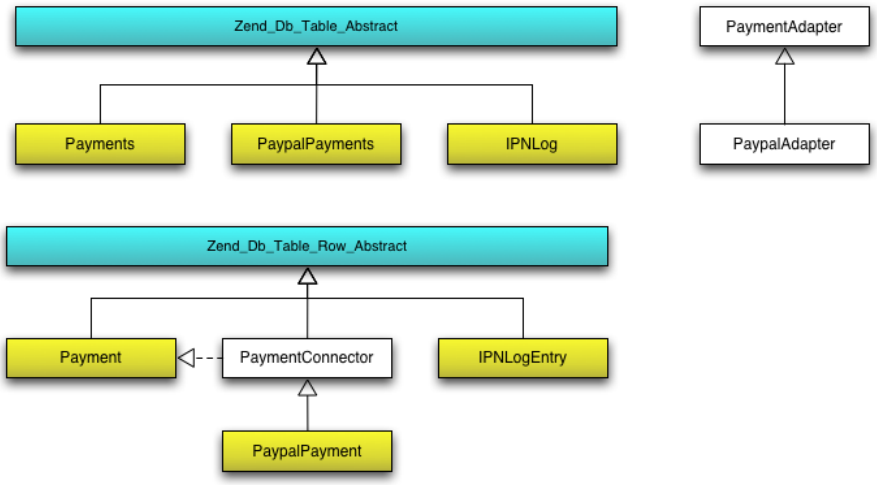

Figure 4.6: The milestone 1 class diagram without functions and properties

The class diagram shows the classes *Payments*, *PaypalPayments* and *IPNLog*. These classes extend the *Zend_Db_Table_Abstract* class, since they represent tables in the database (see section 2.1.1). The *IPNLog* class is created to log all incoming IPN calls and actions of the IPN listener. This information is for logging purposes only.

The table rows are represented by the *Payment*, *PaymentConnector*, *PaypalPayment* and *IPNLogEntry* classes. The connector does not represent a table row but acts like it does because its subclasses represent table rows. The purpose of the connector is explained in section 4.1.1.

The last two classes in the diagram, *PaymentAdapter* and *PaypalAdapter*, deal with communication to the PayPal API. A new payment method could easily be added by extending the *PaymentAdapter* and implement the functions for that specific payment method. More information on the PayPal API and the implementation of the *PaypalAdapter* can be found in sections 2.2 and 5.2, respectively.

4.2.2 Controller and View

In this milestone, two controllers are created for the payment system: *PaymentController* and *PaypalController*. The *PaymentController* takes care of all the action requests and logic needed to interact with the model. The *PaypalController* takes care of all the PayPal specific requests and logic.

For the *PaymentController*, an understanding of the page flow of the user through the payment system is needed to model the controller and views. Such a flow would show the actions a user can take while creating and paying a payment. A simple flow of a user making a payment can be found in Figure 4.7.

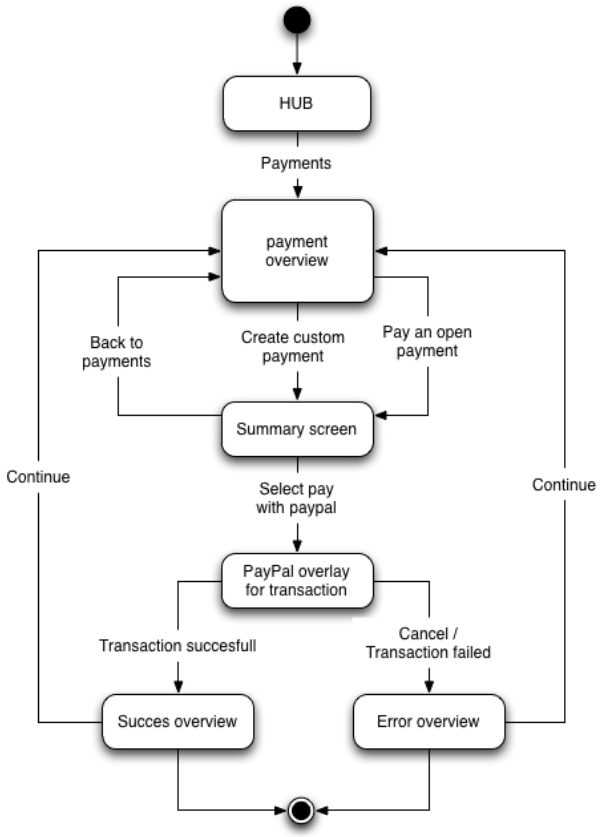

Figure 4.7: A simple page flow diagram of making a payment

Figure 4.7 shows that the *PaymentController* at least needs an overview page and a summary page. More pages seem to be needed if the flow diagram is extended with more detailed information about page navigation, a-synchronous calls (AJAX) and background processes. This extended flow diagram can be found in Appendix G.2. The extended diagram shows that there should also be a page for checking the status of the payment. The *PaypalController* needs to have a success and error action and there should be an action to retrieve the PayPal payment token.

Finally, some standard functionality actions are added. This includes a page to view the details of a payment, a page for deleting a payment and a JavaScript page to hold the JavaScript functions. Saving the newly created payments is handled on the summary page. The *PaypalController* also handles requests for the in section 2.2.4 discussed IPN listener.

The final class diagram for the controllers in this milestone can be found in Appendix G.2.

4.2.3 MVC

Figure 4.8 shows a simple representation of how all the classes and pages have their position in the MVC pattern. The model classes are a representation of the reality, holding all the information provided by the users and the systems of the payment methods (e.g. PayPal). It also takes care of storing this information. The actions in the controller classes manipulate the models data. These actions are, on their turn, activated by external parties like users or the PayPal IPN systems. In the case of the payment system, every action has its own page in the view part of the MVC pattern. This page can be an HTML, JavaScript or JSON file. If needed other view script in other systems can be added to execute a certain action.

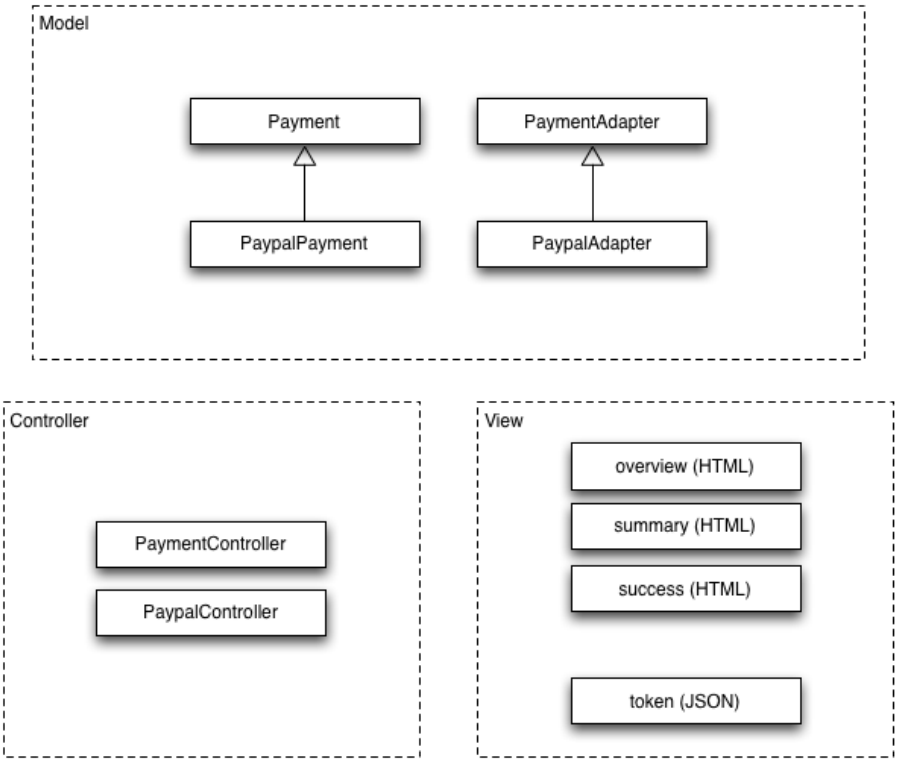

Figure 4.8: Simple view of the payment system in the MVC pattern

4.3 Milestone 2

The functionality to create recurring payments is added in milestone 2. Users do not have to worry about paying rent on time, whether rent should be payed weekly, biweekly, on the first day of the month or the last day of the month.

4.3.1 Changes in the model

The model now has to support the creation of the recurring payments. The recurring payment data is stored in *RecurringPayment* objects. Section 4.1.2 showed that the recurring payment model is constructed in the same way as the *Payment* model itself. These objects are added to the model, which is shown in Figure 4.9. The yellow colored objects are added in milestone 2.

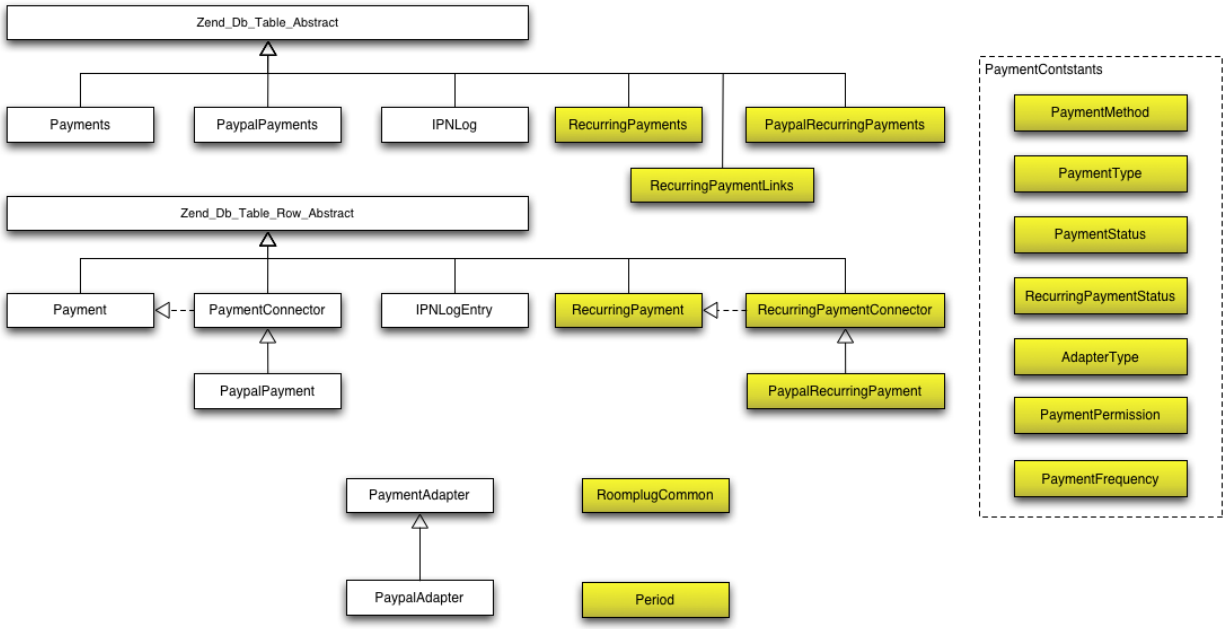

Figure 4.9: The milestone 2 class diagram without functions and properties

In this milestone, constants and functions that are used throughout the payment system and the existing parts of the Roomplug system are put in common classes. This results in seven classes containing the constants, which are packed into one file called *PaymentContants.php*. These classes are shown on the right side of Figure 4.9. There is also a *RoomplugCommon* class containing the functions that are used often.

A recurring payment is only active in a certain period of time. Therefore, the *RecurringPayment* class has a start date and an end date. The Software Improvement Group advised to pack those two dates into one class (see also section 5.3). Therefore, the *Period* class was created.

There are more changes in the already existing classes. Therefore, a detailed class diagram showing the functions and constants in the classes can be found in Appendix G.2.

4.3.2 Changes in the controllers and views

The controllers and views are also updated with new actions and pages to support the recurring payments. The class diagram showing the new controller model can be found in Appendix G.2.

The addition of the *CronjobController* is the biggest change in the diagram. Once a user has pre-approved a set of payments, the Roomplug systems have to take care of executing them on the dates specified by the user. A 'cronjob' or 'crontab' is a UNIX command that is scheduled to run periodically [18]. The command specified on the Roomplug server executes the *recurringpaymentAction* script which creates and executes the in *RecurringPayments* objects specified payments. This command runs twice every day. The second run functions as a backup if the first run fails for some reason. This minimizes the chance that the recurring payments will fail to produce a payment when needed. The *CronjobController* is created to contain all the actions for the scripts that are specified in cronjobs.

4.4 Milestone 3

The goals for milestone 3 changed over time. In the end, it came down to decoupling the payment system and the lease agreement system and to make guest payments possible.

In milestone 1 and 2, the users selected the receiving user based on the lease agreement they have. Over time, the Roomplug business model changed and it became clear that users without a lease agreement on the Roomplug website should also be able to use the payment system. So the connection between the lease agreement system and the payment system was dropped, like shown in Figure 4.10.

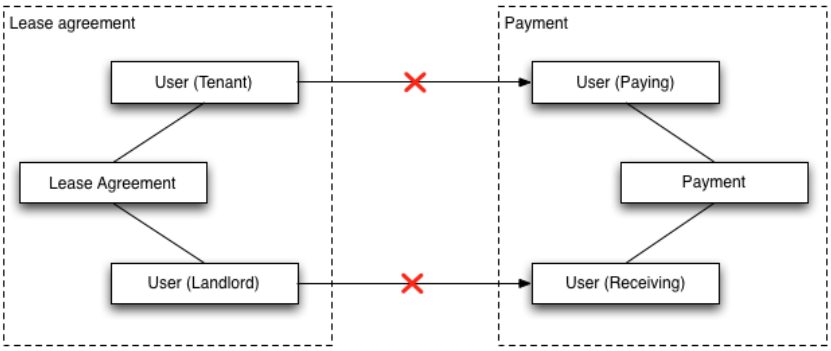

Figure 4.10: Connection between a lease agreement and a payment

The original model of a payment already saved a reference to the receiving and paying user objects itself instead of just referring to a lease agreement. This made the whole change a lot easier since the model would remain untouched. Only the logic in the *PaymentController* and some of the views had to be changed in order to select any user from the Roomplug database.

This also made the implementation of the guest payments a lot easier. In the same brainstorm session to update the Roomplug business model, it also became clear that tenants have a barrier for using the Roomplug rent payment system, because they have to create a profile

before being able to make a payment to their landlord. Roomplug decided that it should also be possible to pay to a Roomplug user without having a profile.

Building the guest payment system was not that big of a change. The payment system already was going to change the way a paying user selects the receiving user. Again, the model remained the same. In the payment system, only the actions and the permissions to execute an action had to change in the *PaymentController* and some views had to be created to let the guest users enter their personal information. The user object in the existing Roomplug system is also changed to support a guest user. Once the guest decides to create an account with the same email address provided while creating the guest payments, the user will have an overview of all the payments executed as a guest.

The changes led to the class diagram of milestone 3, which can be found in Appendix G.2.

4.5 The existing Roomplug system

While developing all the milestones, some features required changes in the already existing Roomplug system. Although it did not require big changes, some models needed additional properties or functions. The properties added to the *User* class is worth mentioning.

A users email address is what separates one PayPal account from another. So in order to pay to a user in the Roomplug system, the email address has to be sent to PayPal. The name is sent with it for verification reasons. Since the name and email address of a user in the Roomplug system can differ from the name and email address that a user entered in his/her PayPal account, specific properties are required for this data. That is why three properties are added to the user object: *paypal_email*, *paypal_first_name* and *paypal_last_name*. These three properties are checked with the PayPal system and if they get verified, the user is all set to receive payments through the payment system using the PayPal payment method.

4.6 User Interface

In general, one of the goals in UI design is to 'keep it simple'. The user interface (UI) for the payment system should align with this goal. Users should not have to go through many screens before their payment finally gets executed and the pages themselves should not be too complex.

Roomplug spent a lot of time on researching and designing its website UI. The payment systems UI is built to support this same style. The Roomplug founders gave some recommendations and improvements on what it should look like and how it should work. They based their recommendations on their knowledge about existing systems in the market. That is why no user input was involved in this project. User experience tests might be done in the future to validate the UI design.

The user payment flow in Figure 4.7 shows the simplicity of the flow of the UI. A user only gets to see three different Roomplug screens on the website to make a payment. The first screen is the payment overview, where a payment can be created. The second screen is the payment summary, showing the payment summary and the different payment methods that can be selected to pay. Finally, there are the success and error screens, depending on the outcome of the payment. Of course, the payment pages of the payment method comes in between the last two screens, but it is not possible to influence that page flow.

The UI design of the pages themselves is kept simple as well. The user manual in Appendix D.1 will walk through all the screens described in the page flow to show the payment system works. This section will only discuss the overview page and the summary page itself to illustrate what the two most important pages look like. These pages are shown in Figure 4.11 and Figure 4.12, respectively.

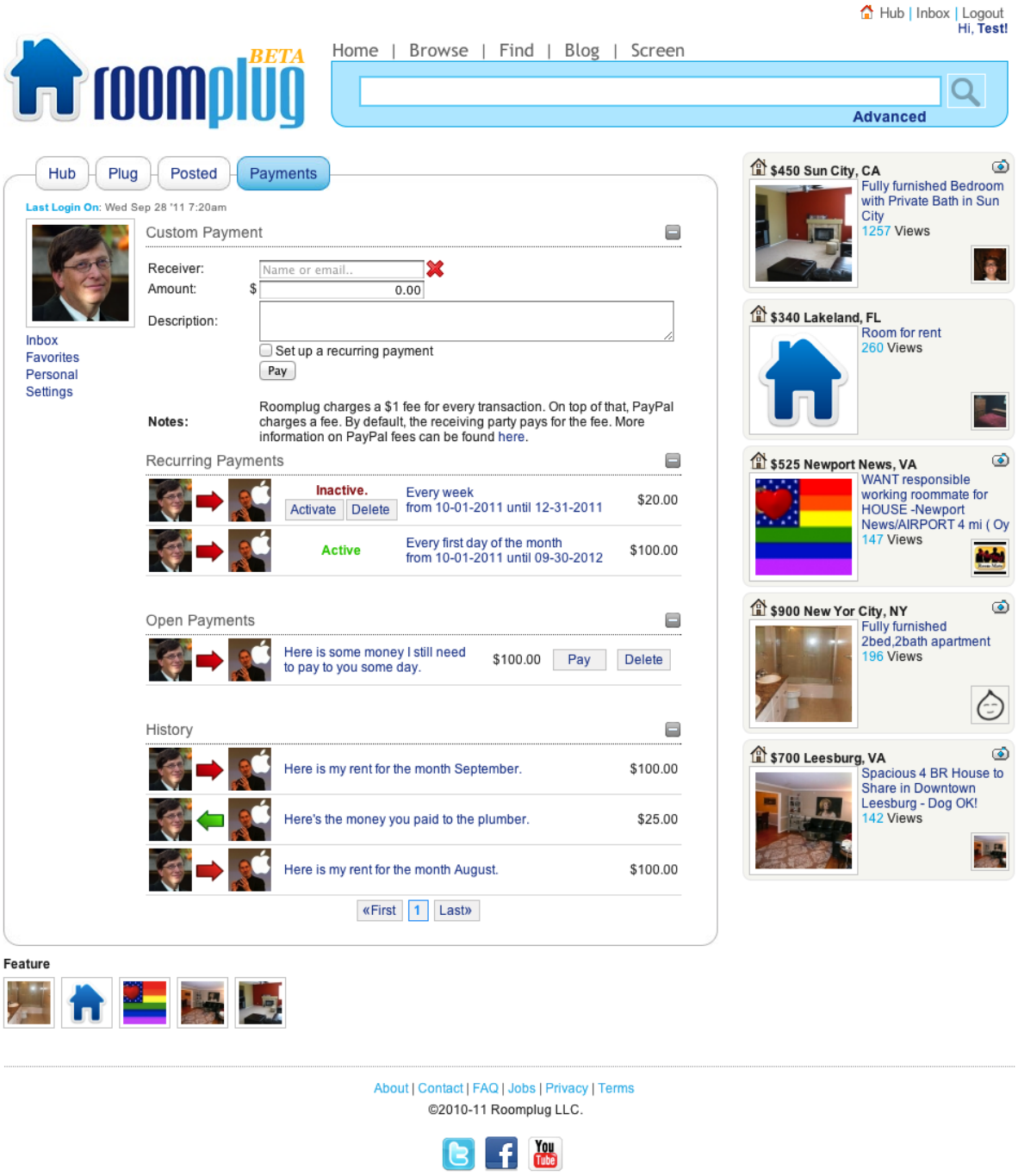

Figure 4.11: The overview page

The overview page is the main entry page to the payment system. From top to bottom, the first part of the payment system can be found under the ‘Custom Payment’ header. Users can directly create a new custom payment using the form. The only thing they have to do is selecting a user, insert an amount and a description and they will be taken to the summary page to select a payment method. The users can check the ‘Set up a recurring payment’ check box if

they want to create a recurring payment. Once they click it, more options will show. Again, more information on how this works can be found in Appendix D.1.

The next header on the page is called ‘Recurring Payments’. This header will only show if there actually are active incoming or any outgoing recurring payments. If the recurring payment is not active, the user gets the option to activate it or to delete it. The meaning of the pictures with the arrow in between them will be discussed in the end of this section, since all the headers have this.

Then there is the ‘Open Payments’ header. This shows all payments that have not been paid yet. The user gets the option to pay or delete such a payment. If the user decides to pay it, they will be taken to the summary page to select their payment method.

Last but not least, there is the ‘History’ header. This will show all the incoming or outgoing payments that have been processed. Only the last ten payments will be shown since the number of payments under this header grows over time. The user can use the page controls at the bottom to view older payments.

And then there is some elements that all headers have in common. Each payment row shows the user images of the two users that participated in the transaction. In between them is either a red or a green arrow. This shows the direction of the money flow and thus what user is the paying user and which user is the receiving user. Red arrows are always pointing to the right and they indicate that the payment is an outgoing transaction. Green arrows always point to the left and they indicate an incoming transaction.

All the describing text on the payments is clickable. Clicking on it will take the user to the details page. The text itself will become underlined and the mouse cursor will change into a pointer once a user hovers over the text. This signals the user that the text is clickable.

Figure 4.12: The (recurring) payment summary page

The summary page is somewhat simpler in comparison to the overview page. It is the second page that users see when they choose to authorize a transaction. This could be paying a payment, or pre-approving a recurring payment. Figure 4.12 shows the overview page which is shown when the user wants to approve a recurring payment. It shows all the relevant information of the recurring payment itself so that the user can check if all the data is correct. The summary page of a custom payment would only show the receiver, the amount and the description.

Then, the user has to select the desired payment method. Buttons for this will show under the (recurring) payment details. Since PayPal is the only method that is supported for now, only that button will show. Clicking it would take the user to the PayPal system screens. Once the user worked their way through them, the Roomplug payment system will show the error or the success page, depending on the outcome of the transaction.

Implementation

In this chapter, samples of code worth notifying, such as our workaround for Zend Framework database classes and the use of the PayPal API, are presented. Next to that, the feedback from the Software Improvement Group (SIG) and the improvements we made because of them are discussed.

5.1 Design workaround Zend Framework

As discussed in section 4.1.1, a workaround had to be used in the model when using the *Zend_Db* classes. The reference map solution described there, is substituted in the solution with a reference to the object itself. In this section examples on how this is actually done are shown.

It starts of with the *rp_payments* table. As shown in Code sample 5.1, a class is added (*Payments*), representing this table and extending the *Zend_Db_Table_Abstract* class, which allows easier access to the actual database. In this code sample, functions are omitted, but those typically are functions to retrieve certain payments from the table. The shown lines define the name of the table in the database, its primary key and the class used to represent a single row from the table.

```
class Payments extends Zend_Db_Table_Abstract {
        protected $_name = 'rp_payments';          //table name
        protected $_primary = 'payment_id';        //primary key in the table
        protected $_rowClass = 'Payment';          //class of the rows returned

        //For this example, redundant functions have been omitted
}
```

Code sample 5.1: The *Payments* class

The *$_rowClass* variable is a reference to the *Payment* class, shown in Code sample 5.2. This class extends *Zend_Db_Table_Row_Abstract*, giving direct access to the fields in the table row. Again, most functions are omitted in this sample, but the *getDescription* and *setDescription* are left in to give an example on how the properties are retrieved and set from the table row.

```
class Payment extends Zend_Db_Table_Row_Abstract {

        //For this example, redundant functions have been omitted

        /**
         * Gets the description of the payment
         * @return string
         */
        public function getDescription() {
                return stripslashes($this->__get("pay_description"));
        }
```

```
        /**
         * Sets the description of the payment
         * @param string $description
         */
        public function setDescription($description) {
                if (is_string($description)) {
                        $this->__set("pay_description", $description);
                }
        }
}
```

Code sample 5.2: The *Payment* class

Now comes the part where a workaround was created for. To store PayPal specific information on a payment, an extra table is created that has a reference to a record in the *Payments* table. Again we create a class representing this table called *PaypalPayments*, as can be seen in Code sample 5.3. The corresponding table is *rp_paypal_payments* with primary key *payment_id*. A single row from this table is represented by the *PaypalPayment* class. Although the table has a foreign key to the *rp_payments* table, the fields of this table are not accessible in the *PaypalPayment* class. This is because the *PaypalPayments* class will only fetch the columns from the table it is representing. The same applies to the *Payments* class and therefor it is not possible to create a class that contains both fields using standard *Zend_Db* classes.

```
class PaypalPayments extends Zend_Db_Table_Abstract {
        protected $_name = 'rp_paypal_payments';                //table name
        protected $_primary = 'payment_id';                     //primary key in the table
        protected $_rowClass = 'PaypalPayment';                 //class of the rows returned

        //For this example, redundant functions have been omitted
}
```

Code sample 5.3: The *PaypalPayments* class

The way this was solved is by creating a class *PaymentConnector*, which extends *Zend_Db_Table_Row_Abstract* and thus represents a table row. By making *PaypalPayment* extend this class, it inherits the *Zend_Db_Table_Row_Abstract* properties and functions. Therefore, the *PaypalPayments* class can use the *PaypalPayment* class to represent one of its rows. To make the *Payment* properties accesible in this class, an instance of a *Payment* object is referenced within the *PaymentConnector*. The reason we do this in the connector and not in the *PaypalPayment* itself, is because we do not want to do this for every payment method that might be added in the future.

In Code sample 5.4 you can see this reference to this *Payment* instance and how the functions of this *Payment* instance are accessed. Code sample 5.5 shows how the *PaypalPayment* class extends the *PaymentConnector*, giving it access to the *Payment* functions and containing only functions for the specific payment method.

```
class PaymentConnector extends Zend_Db_Table_Row_Abstract {
        private $payment = null;                //the payment object where this connects to

        //For this example, redundant functions have been omitted

        /**
         * Gets the description of the payment
         * @return string
         */
        public function getDescription() {
                if($this->invariant())
                        return $this->payment->getDescription();
                else
                        return null;
        }

        /**
         * Sets the description of the payment
         * @param string $description
         */
        public function setDescription($description) {
                if($this->invariant())
                        $this->payment->setDescription($description);
        }
}
```

Code sample 5.4: The *PaymentConnector* class

```
class PaypalPayment extends PaymentConnector {
        //contains PaypalPayment specific functions only
}
```

Code sample 5.5: The *PaypalPayment* class extending *PaymentConnector*

5.2 PayPal API

PayPal provides an SDK for communication with its API. Instead of writing the request in Name-Value Pair format yourself, this SDK lets you set up a request using objects instead [14]. This not only makes it easier to develop, but also makes it more maintainable as it is easier to read.

In the payment system, three kinds of PayPal API operations are used. From the Adaptive Payments API, the *Pay* and *Preapproval* operations are used. From the Adaptive Accounts API, the *GetVerifiedStatus* operation is used. In the next sections, examples are given on how these operations can be invoked using the SDK. Mind that in the code samples shown, code has been left out to highlight the more important properties.

5.2.1 Pay request

Code sample 5.6 gives an overview of properties to set to make a Pay request. Properties worth notifying are the *feesPayer* and the *receiverList*. The *feesPayer* property determines what party pays the PayPal fee. This property can be set to *PRIMARYRECEIVER*, *SECONDARYONLY* and *EACHRECEIVER*. Note that no matter what option is chosen, the total amount of fees stays

the same. With *SECONDARYONLY*, the fees are split among the secondary receivers. In this case, this is Roomplug. Roomplug made the choice that the receiving user should pays the fee, thus the property is set to *PRIMARYRECEIVER*. *EACHRECEIVER* would split the fees among both the primary and secondary receiver, relative to the percentage of the total amount each receives.

The *receiverList* property expects an array with *Receiver* objects. This is also a class provided by the PayPal SDK. Each *Receiver* is identified by its email. The amount to receive and whether the receiver is the primary receiver, is specified for each *Receiver*. When users make a payment, they will only see the primary receiver as the user they send money to.

The *AdaptivePayments* object invokes the actual *Pay* method. The response can result in either a failure or success. If the response was a success, a *payKey* is returned that can be used to redirect the user to the PayPal systems finishing the payment. This is done using AJAX calls and thus this token is returned in JSON format, to make it easier to access.

```
$payRequest = new PayRequest();

//primary receivers (users) pay fee
$payRequest->feesPayer = 'PRIMARYRECEIVER';
$payRequest->memo = $payments[0]->getDescription();

//create Paypal API objects out of the collected receivers
$receiverList = array();

//add roomplug as secondary receiver (as of v0.1.2)
$receiver = new Receiver();
$receiver->email = ROOMPLUG_PAYPAL_EMAIL;
$receiver->amount = ROOMPLUG_FEE/100.0;
$receiver->primary = false;
$receiverList[] = $receiver;

//and the other
foreach ($receivers as $email => $amount) {
        $receiver = new Receiver();
        $receiver->email = $email;
        $receiver->amount = $amount/100.0;
        $receiver->primary = true;
        $receiverList[] = $receiver;
}
$payRequest->receiverList = $receiverList;

// Create service wrapper object
$ap = new AdaptivePayments();

// invoke business method on service wrapper passing in appropriate request params
$response = $ap->Pay($payRequest);
$token = $response->payKey;

//return JSON object
return '{"RESULT":"SUCCESS","TOKEN":"'.$token.'"}';
```

Code sample 5.6: Setting up a *Pay* request

5.2.2 *Preapproval* request

The *Preapproval* request is somewhat similar to the *Pay* request described in section 5.2.1. As Code sample 5.7 shows, no receiver is specified. The *Preapproval* request pre-approves Roomplug to charge the PayPal account a number of times within a period, transferring a specified amount. Roomplug is identified by its personal applicationId. The amount, number of payments and the period are set with the *startingDate*, *endingDate*, *maxNumberOfPayments*, *maxTotalAmountOfAllPayments* and *maxAmountPerPayment* properties.

The *AdaptivePayments* object invokes the actual *Preapproval* method, returning the response, which contains the *preapprovalKey*. This *preapprovalKey* can, once authorized, be used to make payments using the *Pay* request, without user interaction. For this to work, an extra property *preapprovalKey* needs to be set in the *PayRequest* object.

```
//create the request
$preapprovalRequest = new PreapprovalRequest();

//set request params
$preapprovalRequest->startingDate =
        $recurringPayment->getPeriod()->getStartDate()->format('Y-m-d');
$preapprovalRequest->endingDate =
        $recurringPayment->getPeriod()->getEndDate()->format('Y-m-d');
$preapprovalRequest->maxNumberOfPayments = $recurringPayment->getNumberOfPayments();
$preapprovalRequest->maxTotalAmountOfAllPayments =
        $recurringPayment->getTotalAmount()/100.0;
$preapprovalRequest->maxAmountPerPayment = $recurringPayment->getAmount()/100.0;

$preapprovalRequest->memo = $recurringPayment->getDescription();

$ap = new AdaptivePayments();
$response = $ap->Preapproval($preapprovalRequest);

// Check response and return JSON (omitted in this example)
$token = $response->preapprovalKey;

//return JSON object
return '{"RESULT":"SUCCESS","TOKEN":"'.$token.'"}';
```

Code sample 5.7: Setting up a *Preapproval* request

5.2.3 getVerifiedStatus request

This request is somewhat different from the *Pay* and *Preapproval* requests. To begin with, it uses the Adaptive Accounts API instead of the Adaptive Payments API. The *getVerifiedStatus* request is used to verify a users PayPal account. This is done when they set up their PayPal account on Roomplug to enable users to send money to them through the Roomplug systems.

A PayPal account is now identified using the users first and last name, and their email address. The response of the *getVerifiedStatus* request contains information on whether it is a verified account or an unverified account (on success). An example of setting up the *getVerifiedStatus* request, can be seen in Code sample 5.8.

```
//create the request
$VstatusRequest = new GetVerifiedStatusRequest();

$VstatusRequest->emailAddress = $email;
$VstatusRequest->matchCriteria = 'NAME';
$VstatusRequest->firstName = $firstname;
$VstatusRequest->lastName = $lastname;

$aa = new AdaptiveAccounts();
$response=$aa->GetVerifiedStatus($VstatusRequest);

if(strtoupper($aa->isSuccess) == 'FAILURE' ||
        strtoupper($response->accountStatus) == 'UNVERIFIED') {
        //account doesn't exist
        return false;
} else {
        //account should be verified
        if (strtoupper($response->accountStatus) == 'VERIFIED') {
                return true;
        } else {
                //this should never happen
                return false;
        }
}
```

Code sample 5.8: Setting up a *getVerifiedStatus* request

5.3 Software Improvement Group feedback

In this section, the feedback on maintainability, provided by the Software Improvement Group (SIG), will be discussed. The original feedback can be found in Appendix F.1. Since this feedback is written in Dutch, some recommendations are translated and commented on.

The code was sent for review while finishing the second milestone and scored 4 out of 5 stars on maintainability. The unit interface and size were marked as the weak spots. To improve on this, the following feedback was given:

Unit interfacing; Use less parameters for functions.
A high number of parameters indicates lack of abstraction and makes it hard to use the function. Also, the functions are normally longer then average and hard to test.

Unit size; Use smaller methods.
Splitting up long methods into smaller pieces makes them more readable, testable and therefor maintainable.

In the next sections, examples are given on both feedback points and how this was improved. After that, the feedback given on our improvements are discussed.

5.3.1 Examples; Unit interfacing

SIG gave some examples of functions with a high amount of parameters in the code. Those functions and its usage, can be found in Code sample 5.9.

```
private function createRecurringPayment($hash, $amount, $description, $frequency,
                                                                    $startDate, $endDate) {
        //redundant
}

private function recurringInputHasErrors($landlord, $tenant, $hash, $amount, $description,
                                                        $frequency, $startDate, $endDate) {
        //redundant
}

//create a recurring payment old style
$this->createRecurringPayment( $this->getRequest()->getParam('lease_hash'),
                               $this->getRequest()->getParam('pay_amount'),
                               $this->getRequest()->getParam('pay_description'),
                               $this->getRequest()->getParam('rpay_frequency'),
                               $this->getRequest()->getParam('rpay_date_start'),
                               $this->getRequest()->getParam('rpay_date_end')
                             );
```

Code sample 5.9: Example of functions that contain to much parameters

The suggestion made for improvement was to put the variables in a *RecurringPayment* object prior to calling the function. This turned out not to work, since those functions are created to process user input by doing formatting and checks on whether the input is valid. This needs to be done prior to setting the input values in a *RecurringPayment* object, since the setters will not set invalid values.

The way this is solved is by using the *POST* variable from the form submit as the parameter. In the parameter name this is made more clear by calling it *userInput*. Not only makes this more sense in the way the function now looks like, it also makes the use of the function easier as not all the single values have to be fetched as can be seen in Code sample 5.10.

```
private function createRecurringPayment($userInput) {
        //redundant
}

private function recurringInputHasErrors($inputFormatted) {
        //redundant
}

//create a recurring payment new style
$this->createRecurringPayment($this->getRequest()->getParams());
```

Code sample 5.10: Example on how the functions are improved

Another suggestion made, was to combine the *startDate* and *endDate* variables in a *Period* class. This would make the relationship between the two variables clear and save an extra parameter in functions using both dates. Code sample 5.11 shows how the *Period* class is now used and it shows an example of a function having less parameters now because of it.

```
$result['period'] = new Period($startDate, $endDate);

private function checkDates($startDate, $endDate, &$errors) {
        //redundant
}

private function checkPeriod($period, &$errors) {
        //redundant
}
```

Code sample 5.11: The newly created Period class

A different problem encountered with respect to unit interfacing, was the growing amount of actions that had to be added to the *PaymentController*. Basically for every page or AJAX call there is an action in the controller. In an effort to reduce the amount of functions, the decision was made to add a second controller, the *PaypalController*, to handle PayPal specific actions. Not only did this reduce the amount of public functions in the *PaymentController*, it also made a separation between general payment pages and the PayPal pages.

5.3.2 Examples; Unit size

The main remark made about the unit size is that there were files containing a mix of PHP, HTML and JavaScript code. A good example of this, is the summary view page, which is a HTML page containing some JavaScript functions. The JavaScript code was taken out and put it in a separate file. It still needed to be dynamical because of some constants used in it (see Code sample 5.12). If it was stored in a JavaScript file, these constants had to be passed as arguments of the function. This would lead to a large number of arguments. Code sample 5.13 shows how the JavaScript is now included into the summary file.

```
if (obj.STATUS == '<?= PaymentStatus::PAID ?>') {
        //redundant
} else if (obj.STATUS == '<?= PaymentStatus::WAITING ?>' ||
        obj.STATUS == '<?= PaymentStatus::NOT_VERIFIED ?>' ||
        obj.STATUS == '<?= PaymentStatus::OPEN ?>') {
        //redundant
}
```

Code sample 5.12: Constants used in the JavaScript file

```
<!-- Include the JavaScript for the payment process -->
<script type="text/javascript" src="/pay/summaryjs/<?= $paymentId ?>"></script>
```

Code sample 5.13: Importing the dynamic JavaScript file

In the files contianing both HTML and PHP, the PHP is used for different representations of the page only. A minimum amount of logic is done in the HTML pages, but for some files an exception was made. An example is the partial file *payments_list.php*. This view is designed to show a list of payments and uses in multiple scenarios. Since the list should allow paginating, some logic is included in this view, as seen in Code sample 5.14. Also, if there are no payments to show, a message should show rather than presenting an empty table. This is the kind of logic

that is widely seen in view files when coding in PHP; presentation logic rather than business logic [19, 20].

```
if ($this->enablePaginator) {
        //set up paginator and current page items
}

//HTML code

if (count($this->payments) == 0) {
        //print a message rather then showing an empty table
}

foreach ($this->payments as $item) {
        //print a representation of the payment
        if ($this->showActions) {
                //display payment specific action buttons
        }
}

//HTML code

if ($this->enablePaginator) {
        //show the paginator controls
}
```

Code sample 5.14: Enabling paginating in the *payments_list* partial

This partial can thus be used to display payments and allows the programmer to set some options as can be seen in Code sample 5.15.

```
<?php
        //assign payments variable to show the history in its list
        $this->enablePaginator = true;
        $this->payments = $this->history;
        $this->showActions = false;
        print $this->render('/partials/payment/payments_list.php');
?>
```

Code sample 5.15: Using the *payments_list* partial

5.3.3 Feedback on improvements

At the time of writing, this feedback did not get back to us yet.

5.4 Maintainability

Much attention was paid to the maintainability of the code. Zend Framework creates an environment that helps to achieve this. With its MVC structure, code can be easily separated.

The challenge faced with maintainability in this project, is that it needs to be easy to add another payment method, without changing too much of the existing code. Two different parts need to

be taken into account when achieving this: the model and the use of the adapter to execute payments.

The *PaymentConnector* and the other connector classes are introduced in the model, to make it easier to expand it with a new payment method. The only thing that needs to be done is extending the connector classes with classes containing specific information to that payment method. Very little work needs to be done in the existing classes. A weak point on maintainability in this approach, is that when changing or adding a method to the *Payment* class, a corresponding method using this method needs to be added to the *PaymentConnector* class (or a similar class and connector). Even with this weakness, the chosen solution remains the better one. More details on the model design can be found in section 4.1.

In order to keep the execution of payments generic, an abstract class *PaymentAdapter* is created, containing functions commonly used in the payment process. More specifically, those functions are *executePayments*, *setUpRecurringPayment* and *paymentsAreValid*. The *PaypalAdapter* class implements those functions to:

- *executePayments*: Retrieve a token for the given payment from PayPal;
- *setUpRecurringPayment*: Retrieve a pre-approval token for the recurring payment from PayPal;
- *paymentsAreValid:* Checks PayPal specific constraints for a payment.

In the current flow, the *executePayments* and *setUpRecurringPayment* functions are used in AJAX calls from the summary page. Adding a new payment method would involve creating those calls as well. This is built this way, because a different payment method can involve a whole different process. What if, for example, the other method does not work with tokens and redirecting it to an external website at all? A general approach is not possible in this case and therefore those AJAX calls refer to an action in the *PayPalController*, holding PayPal specific actions. For a new payment method, a new controller should be created with actions specific to this new payment method.

It is a different story for the *paymentsAreValid* function. No matter what payment method is used, it can be implemented to return results for a specific payment method in the corresponding adapter created. There is one change that needs to be done when adding a second payment method though.

Currently, the PayPal specific checks are done right after the general input checks and, on error, returned with them. When adding a second payment method this is not the wanted behaviour. PayPal may restrict a certain condition, but a second payment method may not. If the second payment method would allow the payment to be made and PayPal does not, the wanted behaviour is to show the summary page and disable the option to pay with PayPal with the reason why it is disabled. Users can then still choose to use the second option. The place to do the checks can remain the same, the way the response is handled should change.

Testing

In this project, a lot of effort was put in the test suite. This was of great help while extending the system after the core features were build. It also turned out to be extremely useful while refactoring the code.

In this chapter the test plan is presented and the results from testing. The test plan includes what tools we used for testing and what parts had to be tested manually. The results show some examples of bugs that would not have been found manually and how the test reports were used to improve the code.

6.1 Test plan

This section presents the test plan. In the project, a test suite is created right after the first milestone and extended during the development of new features. During the project it became good habit to run the test suite after each new feature or change, thus before each commit.

6.1.1 Automated testing

The automated testing is done using PHPUnit. The target was to get a 100% line coverage for the Controllers and Model classes written. The View classes are indirectly tested by using operations from the Controllers and asserting some pages show, for example, a newly created payment. However, focusing on the coverage of View classes is obsolete since, in the end, the result shown will be determined by the parameters set in the controller.

100% line coverage can be easily achieved by writing tests. Care must be taken not only to get the coverage, but to also validate the output generated by the system when writing them. Not doing this may result in a useless test suite. The reason line coverage is set as the minimum and not, for example, branch coverage, is the feature driven development approach. Since changes in existing functions will most likely occur when implementing new features, a branch coverage requirement would involve analyzing the different branches again and changing the corresponding tests. Too much time would be spent on this process as opposed to checking the line coverage after changes and adding tests for the new lines to cover or adding assertions in the existing tests. This, and the fact that the code coverage report does not analyze branch coverage, gives having line coverage a good balance between time spent and results achieved.

More information on how the PHPUnit test suite is set up, can be read in section 6.2.

6.1.2 Regression testing

Some functionalities are hard to test using PHPUnit. For example, the IPN listener that awaits a call from PayPal to verify a payment was processed, is hard to test using PHPUnit. PayPal provides a tool that enables a developer to send IPN messages to a specified address. This way it is possible to test the IPN listener manually.

Also, the PayPal payment progress, which involves mostly JavaScript code, can not be tested using PHPUnit and therefore needs to be tested manually.

6.2 PHPUnit

Almost every source file in the payment system, has a *mirrored* test file in the test suite. The ones that do not, are tested by using related classes. Such an example is the *Payments* class, which aligns with the *Payment* class. This *mirroring* of files can be seen in the file structure of the Roomplug source files.

The *phpunit.xml* file defines what file is used to bootstrap the Zend Application, where the test suite can be found, and what source files should (or should not) be analyzed on code coverage. An example of the contents can be found in Code sample 6.1.

```
<phpunit bootstrap="./application/bootstrap.php">
    <testsuite name="Unit Tests">
        <directory>./application</directory>
    </testsuite>
    <filter>
        <whitelist>
            <file><!-- Insert path to a source file for code coverage here --></file>
        </whitelist>
    </filter>

    <logging>
        <log type="coverage-html" target="./log/report" charset="UTF-8" yui="true"
hightlight="true"
            lowupperbound="50" highlowerbound="80" />
    </logging>
</phpunit>
```

Code sample 6.1: An example of the *phpunit.xml* file

The bootstrap file referenced is somewhat different from the system's bootstrap file. It defines the environment to be the testing environment, which allows a different database to be used than the development environment. Also, *Zend_Controller_Front* will not be used to dispatch pages, as dispatching pages is done manually in tests.

To utilize Zends libraries, a *TestCase* class is created and extends *Zend_Test_PHPUnit_ControllerTestCase*. This gives access to Zend specific assertions and methods to make requests. Each other test class extends the *TestCase* class, giving it the same functionalities.

Beside general set up functionalities implemented in the *TestCase* class, it also implements the *setUpBeforeClass* function, which PHPUnit automatically calls before starting test runs on a new class file. In this function, the test database is cleared and two users that are used in the tests are added to the user database, as can be seen in Code sample 6.2. This ensures the environment is clean when starting tests in a certain test class.

```
class TestCase extends Zend_Test_PHPUnit_ControllerTestCase {

	public static function setUpBeforeClass() {
		$db = Zend_Registry::get('db');
		//clear database
		//add two users
	}

}
```

Code sample 6.2: The *TestCase* class providing a clean environment

In a test class itself, the environment needs to be kept clean between every test function. An example of the *setUp* and *tearDown* functions of the *PaymentTest* class can be found in Code sample 6.3. The *setUp* function is called before every test function starts, the *tearDown* is called after each test function ends. This allows the programmer to make sure the environment is clean before each test run.

```
class PaymentTest extends TestCase {
	private $payments = null;
	private $payment = null;
	private $users = null;
	private $user = null;

	public function setUp() {
		//set up objects
		$this->payments = new Payments();
		$this->payment = $this->payments->createRow();

		$this->users = new Users();
		$this->user = $this->users->createRow();
	}

	public function tearDown() {
		//clear objects
		if ($this->payment != null)
			$this->payment->delete();
		if ($this->user != null)
			$this->user->delete();

		unset($this->payments);
		unset($this->payment);
		unset($this->users);
		unset($this->user);
	}
}
```

Code sample 6.3: The *setUp* and *tearDown* classes, provided to clean up your environment

Sample code showing how Zends provided functions can be used to dispatch a page and check the content of the resulting page, can be found in Code sample 6.4. Here the *POST* variables are set, that are used when creating a recurring payment. After dispatching the summary page, the recurring payment should be set up, with the information given in the *POST* variables. Also, the controller and action used are checked in every test.

```
public function testSummaryActionRecurringWeekly() {
        $startDate = $this->dateCurrent;
        $endDate = $this->dateTenWeeks;

        //set the POST variables for the next request
        $this->request->setMethod('POST')->setPost(array(
                                        'pay_receiver_id' => '484',
                                        'pay_amount' => '100.00',
                                        'pay_description' => 'PHPUnit test payment',
                                        'pay_recurring' => 'pay_recurring',
                                        'rpay_frequency' => PaymentFrequency::WEEKLY,
                                        'rpay_date_start' => $startDate->format('m/d/Y'),
                                        'rpay_date_end' => $endDate->format('m/d/Y')
                                ));

        //make the request
        $this->dispatch('/pay/summary');

        //assert the information is shown
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'User, Test');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '$100.00');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'PHPUnit test payment');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'Every week');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', $startDate->format('m-d-
Y'));
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', $endDate->format('m-d-
Y'));
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '10');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '$100.00');

        //assert the right actions were taken
        $this->assertController('payment');
        $this->assertAction('summary');
        $this->assertModule('main');
        $this->assertResponseCode(200);
}
```

Code sample 6.4: A test using Zend Framework specific assertions

6.3 Results

In this chapter the various benefits and pains that were encountered while testing are discussed. The examples show bugs caught because of automated testing, code analysis results and how those results were used to improve code are discussed and room there is for improvement is discussed.

6.3.1 Pros vs Cons

Having a big test suite has both its advantages and its disadvantages. For example, the time put into creating the tests is relatively high compared to the time spent on creating the system itself. This is a trade off made resulting in bugs being more likely, or less likely to slip into the code.

Another disadvantage of having a big test suite, is the time it takes to run. In Figure 6.1, a screen dump of a single run can be seen. It took 3:37 minutes to run. This time is way too long

to run the test suite between each change made to the code. Therefore, test were only run after some manual testing and before each commit.

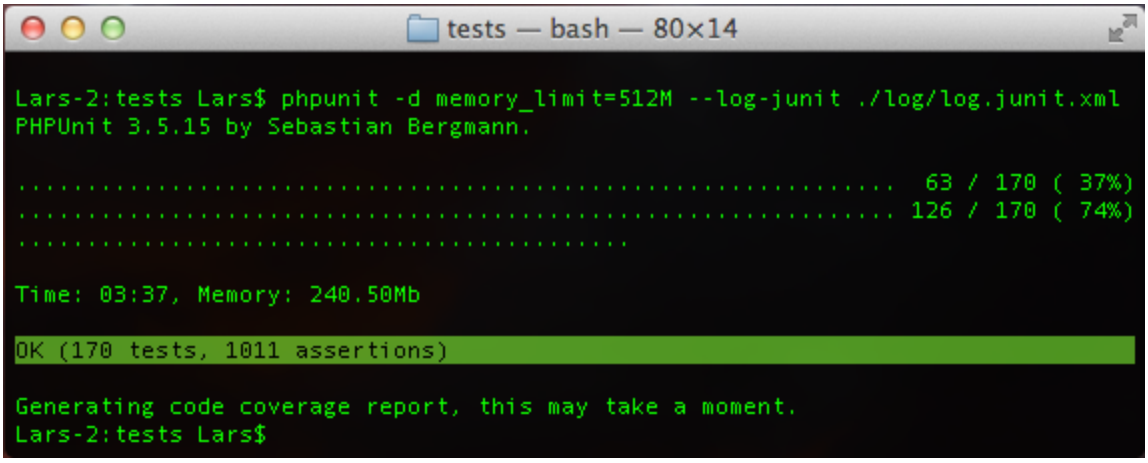
tests — bash — 80×14
Lars-2:tests Lars$ phpunit -d memory_limit=512M --log-junit ./log/log.junit.xml
PHPUnit 3.5.15 by Sebastian Bergmann.
63 / 170 (37%)
126 / 170 (74%)
Time: 03:37, Memory: 240.50Mb
OK (170 tests, 1011 assertions)
Generating code coverage report, this may take a moment.
Lars-2:tests Lars$

Figure 6.1: A screenshot of a test suite run

6.3.1.1 Finding bugs

Although running the complete test suite took quite some time, various bugs were found because of it. A good example of a bug that would be hard to find by manually testing, but was found from writing tests, is shown in Code sample 6.5. Here one of the possible statuses of a *Payment* was forgotten and could therefore not be set into the status property.

```
/**
 * Checks for a valid value for the status field
 * @param string $status
 * @return bool
 */
private function checkStatus($status){
        //status should be one of the following
        switch ($status){
                case 'approved':
                case 'not approved':
                case 'deleted':
                case 'waiting for': //forgotten status
                        return true;
                default:
                        return false;
        }
}
```

Code sample 6.5: An example of introducing an error by forgetting a minor thing

Also, a wrongly formatted JSON object was found because of testing. In Code sample 6.6, the creation of the JSON object can be found, using wrong parenthesis.

```
//format JSON object (before)
$JSon = "{'RESULT':'FAILURE','ERROR':";
//format JSON object (after)
$JSon = '{"RESULT":"FAILURE","ERROR":';

//return JSON object (right format)
return '{"RESULT":"SUCCESS","TOKEN":"'.$token.'"}';
```

Code sample 6.6: An example on how an error was found in a JSON format

6.3.1.2 Refactoring

Refactoring code is another example of when having an extensive test suite, is an advantage. The code sample shown in Code sample 6.5 was refactored in a later stadium to use constants instead of strings. This refactoring involved replacing all the string instances with the constant. How this was done for the *checkStatus* function, can be seen in Code sample 6.7. This change makes it easy to change the underlying data representation (for example, replace the strings by integers in the database). By changing the data representation, the test suite will show the places where the original statuses used, were missed.

```
/**
 * Checks for a valid value for the status field
 * @param string $status
 * @return bool
 */
private function checkStatus($status){
        switch ($status){
                case RecurringPaymentStatus::APPROVED:
                case RecurringPaymentStatus::NOT_APPROVED:
                case RecurringPaymentStatus::DELETED:
                case RecurringPaymentStatus::WAITING:
                        return true;
                default:
                        return false;
        }
}
```

Code sample 6.7: The refactored version of *checkStatus*

6.3.1.3 Dependency on PayPal

One of the problems encountered with the current test suite, is its dependency on the PayPal systems. Certain tests require a connection to the PayPal systems. For example, to initiate a payment, a token is retrieved from the PayPal systems to use when redirecting to the PayPal payment flow. If the Internet connection drops, or when PayPal has down time, the tests will fail. There is no real fix for this, it's up to the programmer to identify if the problem lies in the PayPal system or not.

6.3.2 Coverage and complexity reports

PHPunit includes the *PHP_CodeCoverage* library, generating a code coverage report and statistics report about complexity of the code analyzed after each test run. Using these reports, the code and tests were enhanced as to make complexity lower and code coverage higher.

The index used for a functions complexity, is the Change Risk Analysis and Predictions (CRAP) index. This index represents the cyclomatic complexity versus the code coverage of the function. This cyclomatic complexity indicates the number of different paths a program can take through a function. *"The C.R.A.P. (Change Risk Analysis and Predictions) index is designed to analyze and predict the amount of effort, pain, and time required to maintain an existing body of code."* [21]. A lower CRAP score means it is less complex and well covered by tests. Determining what a good CRAP score is, is up to the programmer to decide. Often, a threshold of 30 is suggested, but while developing the payment system it became clear a threshold of 10 was also a reachable target.

In Figure 6.2, the CRAP score is shown in a table with cyclomatic complexity versus code coverage. Since 100% code coverage is achieved on all functions, a maximum score of 10 implies having a function with a cyclomatic complexity of 10. The specific function having a CRAP index of 10, is the *prepareView* function in the *PaymentController*. This function involves a switch statement, taking different actions on the action requested.

		Code Coverage in %																				
		100	95	90	85	80	75	70	65	60	55	50	45	40	35	30	25	20	15	10	5	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	4	4	4	5	5	6
C	3	3	3	3	3	3	3	3	3	4	4	4	4	5	5	6	7	8	9	10	11	12
y	4	4	4	4	4	4	4	4	5	5	5	6	7	7	8	9	11	12	14	16	18	20
c	5	5	5	5	5	5	5	6	6	7	7	8	9	10	12	14	16	18	20	23	26	30
l	6	6	6	6	6	6	7	7	8	8	9	11	12	14	16	18	21	24	28	32	37	42
o	7	7	7	7	7	7	8	8	9	10	11	13	15	18	20	24	28	32	37	43	49	56
m	8	8	8	8	8	9	9	10	11	12	14	16	19	22	26	30	35	41	47	55	63	72
a	9	9	9	9	9	10	10	11	12	14	16	19	22	26	31	37	43	50	59	68	78	90
t	10	10	10	10	10	11	12	13	14	16	19	23	27	32	37	44	52	61	71	83	96	110
i	11	11	11	11	11	12	13	14	16	19	22	26	31	37	44	53	62	73	85	99	115	132
c	12	12	12	12	12	13	14	16	18	21	25	30	36	43	52	61	73	86	100	117	135	156
	13	13	13	13	14	14	16	18	20	24	28	34	41	50	59	71	84	100	117	136	158	182
C	14	14	14	14	15	16	17	19	22	27	32	39	47	56	68	81	97	114	134	157	182	210
o	15	15	15	15	16	17	19	21	25	29	36	43	52	64	77	92	110	130	153	179	208	240
m	16	16	16	16	17	18	20	23	27	32	39	48	59	71	86	104	124	147	173	203	235	272
p	17	17	17	17	18	19	22	25	29	35	43	53	65	79	96	116	139	165	194	228	265	306
l	18	18	18	18	19	21	23	27	32	39	48	59	72	88	107	129	155	184	217	254	296	342
e	19	19	19	19	20	22	25	29	34	42	52	64	79	97	118	143	171	204	241	282	329	380
x	20	20	20	20	21	23	26	31	37	46	56	70	87	106	130	157	189	225	266	312	363	420
i	21	21	21	21	22	25	28	33	40	49	61	76	94	116	142	172	207	247	292	342	399	462
t	22	22	22	22	24	26	30	35	43	53	66	83	103	127	155	188	226	270	319	375	437	506
y	23	23	23	24	25	27	31	37	46	57	71	89	111	137	168	204	246	294	348	409	477	552
	24	24	24	25	26	29	33	40	49	61	76	96	120	148	182	222	267	319	378	444	518	600
	25	25	25	26	27	30	35	42	52	65	82	103	129	160	197	239	289	345	409	481	561	650
	26	26	26	27	28	31	37	44	55	69	88	111	138	172	212	258	311	372	441	519	606	702
	27	27	27	28	29	33	38	47	58	74	93	118	148	184	227	277	335	400	475	558	652	756
	28	28	28	29	31	34	40	49	62	78	99	126	158	197	243	297	359	429	509	600	700	812
	29	29	29	30	32	36	42	52	65	83	106	134	169	211	260	317	384	460	545	642	750	870
	30	30	30	31	33	37	44	54	69	88	112	143	180	224	277	339	410	491	583	686	802	930

Figure 6.2: Cyclomatic complexity versus code coverage [22]

6.4 Future test automation

Although an extensive test suite was created during the development of the payment system, it does not stop expanding. For each feature to be added in the future, new tests related to this new feature have to be added. Also, changes in the current code that modify the output on a

page, may involve changing the assertions done in the existing test cases. Basically, changing or extending the current payment system implies changing the test suite as well.

Automated test work that was not done in this project is GUI testing. Tools like Selenium IDE [23] could be used to create a test suite involving actual user actions to be executed and again asserting the pages show what they should show. This could help in creating tests that check for true/false paths in the work flows of the payment process. This has to be done manually now.

Conclusion

This chapter will revisit the problem defined in chapter 1. It uses chapters 2 through 6 to draw conclusions on how and if the problem has been solved and what the added value of the system is to Roomplug. After that, recommendations and future work will be discussed.

7.1 Results

This project was about the problem formulated in this question: “How can the current payment process between renters and landlords be improved by using the tools and services of the Roomplug website?”. This problem is solved by creating a way for users to pay their rent online, using the Roomplug website. The solution meets all the requirements stated in the problem definition.

1. *Roomplug wants a prototype of a rent payment system, which allows renters and landlords to pay each other online.*

The prototype of the online payment system is completed and is currently active on the live beta site of Roomplug. Renters now have the possibility to pay their landlord through the Roomplug website.

2. *The payment system should be designed and build to be a stand alone system, using data from other components of the Roomplug website.*

The payment system is, like the rest of the Roomplug website, created using the MVC pattern in Zend Framework. All components of the payment system could be transferred and used in another system using this same configuration. The payment system only requires the *User* class to be present and the system needs to adopt the new sites security methods.

3. *Roomplug users should be able to receive payments from everyone, even if the paying person does not have a Roomplug account.*

Guest payments have been implemented requiring only the receiving user to have a Roomplug account. Paying users can go to the guest payment page and create a payment to a Roomplug user there. Once they create an account, all the payments that the user has made as a guest, will show on his/her history page provided they entered the email address of their new Roomplug account on every payment.

4. *Roomplug users should be able to set up recurring payments to pay their rent automatically.*

Users can set up recurring payments and schedule them to occur during a certain period. Next to that, they can choose four different frequencies of payment: weekly, biweekly, every first day

of the month and every last day of the month. The Roomplug server runs a cronjob twice a day. The second run functions as a backup if the first run fails for some reason. This minimizes the chance that the recurring payments will fail to produce a payment when needed.

5. *Roomplug users should be able to manage their payments. This managing could include, for example, an overview of the payment history.*

The Roomplug users can see all the incoming and outgoing payments executed in the past on the overview page. They can also see their outgoing open payments, their active or inactive outgoing recurring payments, and the active incoming recurring payments. The users can also click on the description of the payment to see the payment details. Users can delete an outgoing payment if the execution of that payment has not started yet. Outgoing recurring payments can be de-activated when active or deleted if not active.

6. *The online rent payment system should be designed to support multiple payment methods. PayPal is the method that is first to be adopted by the online payment system.*

The payment system is designed in such a way that other payment methods could be added easily. A guide on how to do this can be found in Appendix D. Currently, both custom payments as recurring payments can be paid or pre-approved using PayPal as a payment method. Users can set up their PayPal account data in their Roomplug profile. They are ready to receive payments through PayPal as soon as they did this.

All together, this project has brought a fully functional prototype of an online rent payment system to the Roomplug website. This came with elaborate documentation and manuals to maintain and expand the payment system in the future. Documentation includes model design documents, class diagrams, and flow diagrams. Manuals include the user manual, the maintenance manual, and the test manual.

FDD proved to be the right methodology for this project in solving the problem. During the project, Roomplug was still trying to refine its business model and they watched the competition closely. This caused the requirements and milestones to change over time. The features that did not make it into this project can still be implemented in the future, if wanted. This will be discussed in section 7.2.

Although the design work is created for the payment system, lessons are learned from the design structure. Using Zend Framework and the MVC pattern required some workarounds to prevent implementation problems or to prevent the whole model to become too complex. Future development on the Roomplug system or any other Zend Framework MVC system can benefit from this knowledge.

The elaborate PHPUnit test classes that have been written after each milestone, have been of great value during the development of new milestones and changes that had to be made in older ones. It adds up to the maintainability and expandability of the system as a whole.

From the beginning of the project, Roomplug thought of the payment system as a critical tool for the Roomplug website. The system is now live on the beta site and it has to prove its value.

Roomplug believes the future of paying rent will be by means of an online platform. By utilizing the Paypal API and their adaptive payment system, our software development ninjas, Dirk & Lars, have masterfully crafted the art for us to generate income at a $1 fee for each rental transaction through this innovative process. We are concurrently in the works with Paypal to be invited into their Business Pilot program in which the transaction fee on their part will be reduced down from 2.9% + $0.30 on every transaction to $0.50. With this new fee structure and partnership in place, the overall transaction cost is dramatically reduced, and hence drive Roomplug business model to help our customers to socially rent and collect payment on our site.

We commend the efforts of Dirk Guijt and Lars Tijhuis both, our Dutch engineers, for building this platform, which will greatly impact the landscape and future of renting online! Roomplug is truly grateful and thank them for their energies, knowledge, workmanship, and collaboration in providing us an innovative service to the rental market space. We hope to be the premier leader in this domain for many users to "Socially Rent and Live Happily Under One Roof!"

- Rodney Nguyen, Founder & CEO of Roomplug

7.2 Recommendations

This section will discuss some recommendations and future work on the Roomplug system.

7.2.1 Features

Unfortunately, this project could not cover all features that had been thought of. Roomplug made some choices to prioritize features. Some of them did make it into a milestone but some did not. Payment requests, payment suggestions and pre-filled payments are three of the features that were in the original requirements but were moved out of this project because other features supported the business model better.

The request payment feature should give renters or landlords the possibility to request a payment from one another. For example, if the tenant paid for the plumber, he or she can request a reimbursement of the costs from the landlord. The landlord can then follow up or deny that request with the click of a button. This can be an important feature when Roomplug implements more and more tools for landlords to manage their property.

Payment suggestions use the information in the lease agreement to make suggestions about payments that should be paid or requested (if requests are implemented). This can help reminding the renters to pay their rent on time if they did not set up a recurring payment for it. This feature needs to have the information of the lease agreement to work. It will only be useful to implement this if the lease agreement system is being actively used.

Pre-filled payments let users create a payment without entering any data. Instead, the information from their lease agreement is used. They do not have to enter the data themselves. Like the payment suggestions, this feature only works once there actually is a lease agreement in the Roomplug system.

7.2.2 Future work

Although the Roomplug system came a long way during this project and before, there is still a lot of room for improvement. This does not only include adding the features mentioned above. Before work on the payment system started, Roomplug wanted to integrate all the tools with each other. During the project, it became clear that a structure in which all the tools are stand alone systems, optionally using each others data, is better. Luckily, the payment systems design already supported this new business structure.

The marketplace that Roomplug provides does not have such a design. One of the big problems is that the system lacks a 'property' object holding information on the property that is rented. All that information is now stored in an 'advertisement' object. Once there is a lease agreement, the landlord can delete the advertisement and all information on the property is lost. It is recommended to separate the property and the advertisement into two separate objects, since the property information can then be used in other parts of the website, like the lease agreement system or when searching for the receiving user of a payment.

The lease agreement itself is stored as plain text, so changing the lease agreement template will not affect it. This does present another problem. The other parts of the Roomplug website, including the payment system, cannot easily use the information from the lease agreement, since the relevant information is not stored in separate database fields. There is just one field holding the complete text. Although storing the lease agreement as a whole is good practice, it is recommended to store the relevant information of the lease agreement in separate fields as well, so other parts of the website can use it.

The payment systems UI design is based on the Roomplug founders experience and knowledge of other online payment systems. It would however be recommended to do some user experience tests with real-life users to test how the UI works out for them and maybe improve the UI based on their experiences with it.

Improvements are also to be made in the implementation of the UI in Zend Framework. Right now, all the Roomplug views have the whole HTML and CSS implementation of the UI design in them. Allmost all of this code is copied over and over again in the views. All these views have to be updated if there is some big UI change. The payment system solved this problem by using the *Zend_Layout* library discussed in section 2.1.2. It is recommended to use this functionality throughout the whole website. This will improve the UI maintainability dramatically.

The last recommendation involves test code. Although the payment system got elaborate test classes, all other parts of the website have none at all. It is recommended to create tests for each feature in order to make future development throughout the Roomplug website easier and

safer. Having test code was critical in the development of the payment system. Also, there are no automated GUI tests in the current test suite. This could be done in the future as well.

References

[1] “About Roomplug”, Roomplug. Visited on July 27, 2011. <http://www.roomplug.com/about>.

[2] “Credit Report, Credit Score, Credit Check – TransUnion.com”, TransUnion. Visited on Sept 30, 2011. <http://www.transunion.com>.

[3] “Feature Driven Development (FDD) and Agile Modeling”, Ambler, S.W. Visited on July 29, 2011 <http://www.agilemodeling.com/essays/fdd.htm>.

[4] “Zend Framework components”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/components>.

[5] “Zend Db”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.db.html>.

[6] “Zend Db Adapter”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.db.adapter.html>.

[7] “Zend Db Table”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.db.table.html>.

[8] “Zend Db Select”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.db.select.html>.

[9] “Zend Db Table Row”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.db.table.row.html>.

[10] “Zend Layout”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.layout.introduction.html>.

[11] “Zend Framework and MVC introduction”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/learning.quickstart.intro.html>.

[12] “Zend View”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.view.introduction.html>.

[13] “Zend Controller”, Zend Technologies Ltd. Visited on October 4, 2011. <http://framework.zend.com/manual/en/zend.controller.quickstart.html>.

[14] “Adaptive Payments Developer Guide”, PayPal, Inc. Document Number:10097.en_US-20110427. Visited on August 8, 2011. <https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_AdaptivePayments.pdf>

[15] “PHPUnit manual”, Bergmann, S. Visited on October 3, 2011. <http://www.phpunit.de/manual/3.6/en/phpunit-book.pdf>

[16] “PHP_CodeCoverage”, Bergmann, S. Visited on October 3, 2011.
<https://github.com/sebastianbergmann/php-code-coverage>

[17] “Zend Test PHPUnit”, Zend Technologies Ltd. Visited on October 3, 2011.
<http://framework.zend.com/manual/en/zend.test.phpunit.html>

[18] “Crontab”, The IEEE and The Open Group. Visited on October 1, 2011.
<http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html>.

[19] “Template engines”, Lozier, B. Visited on October 8, 2011.
<http://www.massassi.com/php/articles/template_engines/>.

[20] “All about Smarty”, New Digital Group, Inc. Visited on October 8, 2011.
<http://www.smarty.net/about_smarty>.

[21] “Pardon My French, But This Code Is C.R.A.P. (2)”, Savoia, A. Visited on October 3, 2011.
<http://www.artima.com/weblogs/viewpost.jsp?thread=210575>

[22] “PHP bekommt frischen CRAP”, Dusch, V. Visited on October 3, 2011.
<http://www.phphatesme.com/blog/softwaretechnik/php-bekommt-frischen-crap/>

[23] “Selenium HQ”, Selenium. Visited on October 3, 2011. <http://seleniumhq.org/>

Definitions and Abbreviations

AJAX	Abbreviation for 'a-synchronous JavaScript and XML'. AJAX calls are generally used to load data dynamically on a website without refreshing the page.
API	Abbreviation for 'application programming interface'. The API of a system lets developers communicate to that system in a controlled fashion.
Bootstrap file	The file that is used to initiate the Zend Application with the right configuration.
C.R.A.P.	Abbreviation for 'change risk analysis and predictions'. The CRAP score is a benchmark to measure code complexity versus code coverage by tests.
Cronjob (or crontab)	A cronjob or crontab is a scheduled server task, which executes a defined command.
CSS	Abbreviation for 'cascading style sheet'. CSS is a markup language to describe the presentation semantics of a document.
FDD	Abbreviation for 'feature driven development'. FDD is an agile software development methodology using features and milestones to do incremental implementation.
Feature	A distinct functionality in the software.
GUI	Abbreviation for 'graphical user interface'. See also UI.
HTML	Abbreviation for 'HyperText Markup Language'. The language used to provide the structure for a web page.
IPN	Abbreviation for 'instant payment notification'. A service provided by PayPal to notify external systems of transactions executed in that systems corresponding PayPal application.
JavaScript	A client based script language used in website development.
JSON	Abbreviation for 'JavaScript object notation'. Data formatting convention, can be used to transfer data between different script or programming languages.
Milestone	Set of features. A milestone is also a point in a development track, marking the completion of a set of features.
MVC	Abbreviation for 'model-view-controller'. MVC is an architectural pattern used in software engineering.

Partial	A reusable piece of code, normally used on multiple places.
Payments	
incoming payment	A payment that a user received.
outgoing payment	A payment that a user paid.
open payment	A payment that was created, but not yet executed.
custom payment	A single payment of which the details are defined by the user.
recurring payment	A payment, scheduled to automatically execute with a given frequency in a defined period.
pre-approved payment	A payment that a user gave permission for, prior to its execution. Normally an occurrence of a recurring payment.
PayPal	PayPal is an e-commerce business allowing payments and money transfers to be made through the Internet.
SIG	Abbreviation for 'Software Improvement Group'. A consultancy firm that provides assessments of risks related to software systems.
SDK	Abbreviation for 'software development kit'. A set of development tools that allows for the creation of applications for a certain software package.
Test suite	A set of tests with the goal to test a certain system.
UI	Abbreviation for 'user interface'. The front end of a system, gives users the possibility to interact with it.
Users	
user	A person using the system. May or may not have registered with the Roomplug system.
Roomplug user	A person that has registered with the Roomplug system.
guest user	A person using the payment system without having an account on Roomplug.

Appendices

Appendix A: List of figures 71
Appendix B: List of code samples 73
Appendix C: Preliminary Report 75
Appendix D: Manuals 103
D.1 User manual 103
D.2 Maintenance manual 115
D.3 Test manual 116
Appendix E: Roomplug 121
E.1 Roomplug 121
E.2 Payment System 121
E.3 Startup Company 122
E.4 USA 122
References 122
Appendix F: Software Improvement Group feedback 123
F.1 Initial feedback 123
F.2 Feedback on improvements 124
Appendix G: Digital documents 125
G.1 Source code 125
G.2 Extended design documents 125
G.3 phpDocuments 125
References 125

Appendix A: List of figures

Figure 1.1: The time schedule for the development of the payment system 11
Figure 2.1: The abstract structure for a Chained Payment [14] 15
Figure 2.2: The flow for executing a payment through PayPal [14] 16
Figure 2.3: Executing a pre-approved payment [14] 17
Figure 2.4: The IPN flow 18
Figure 4.1: Inheritance and the database mapping problem (and solution) 26
Figure 4.2: The basic model of the payment objects 27
Figure 4.3: The basic model of the payment objects using Zend Framework 27
Figure 4.4: The payment system model 28
Figure 4.5: The relational model of the payment system 29
Figure 4.6: The milestone 1 class diagram without functions and properties 30
Figure 4.7: A simple page flow diagram of making a payment 31
Figure 4.8: Simple view of the payment system in the MVC pattern 32
Figure 4.9: The milestone 2 class diagram without functions and properties 33
Figure 4.10: Connection between a lease agreement and a payment 34
Figure 4.11: The overview page 37
Figure 4.12: The (recurring) payment summary page 39
Figure 6.1: A screenshot of a test suite run 55
Figure 6.2: Cyclomatic complexity versus code coverage [22] 57

Appendix B: List of code samples

Code sample 5.1: The *Payments* class 41
Code sample 5.2: The *Payment* class 41
Code sample 5.3: The *PaypalPayments* class 42
Code sample 5.4: The *PaymentConnector* class 43
Code sample 5.5: The *PaypalPayment* class extending *PaymentConnector* 43
Code sample 5.6: Setting up a *Pay* request 44
Code sample 5.7: Setting up a *Preapproval* request 45
Code sample 5.8: Setting up a *getVerifiedStatus* request 46
Code sample 5.9: Example of functions that contain to much parameters 47
Code sample 5.10: Example on how the functions are improved 47
Code sample 5.11: The newly created Period class 48
Code sample 5.12: Constants used in the JavaScript file 48
Code sample 5.13: Importing the dynamic JavaScript file 48
Code sample 5.14: Enabling paginating in the *payments_list* partial 49
Code sample 5.15: Using the *payments_list* partial 49
Code sample 6.1: An example of the *phpunit.xml* file 52
Code sample 6.2: The *TestCase* class providing a clean environment 53
Code sample 6.3: The *setUp* and *tearDown* classes, provided to clean up your environment 53
Code sample 6.4: A test using Zend Framework specific assertions 54
Code sample 6.5: An example of introducing an error by forgetting a minor thing 55
Code sample 6.6: An example on how an error was found in a JSON format 56
Code sample 6.7: The refactored version of *checkStatus* 56

Appendix C: Preliminary Report

Before work on design and implementation started, a preliminary report was created. This report can be found in this appendix.

Preliminary Report

Online Rent Payment System

Version 0.3 **August 1, 2011**

Delft University of Technology
Faculty EEMCS

Bachelors Project Group (IN3405)
Dirk Guijt 1308084
Lars Tijhuis 1308335

Thesis Committee

Chairman:	Prof. Drs. Dr. L.J.M. Rothkrantz
B.Sc. Coordinator:	Drs. P.R. van Nieuwenhuizen
Company Supervisors:	R. Nguyen, S. Kanajan

Table of Contents

Table of Contents 79
1. Introduction 81
  1.1 Roomplug 81
2. Project Description 83
  2.1 Goal 83
  2.2 Current system 83
    2.2.1 Users and advertisements 83
    2.2.2 Searching advertisements 84
    2.2.3 Application and Lease agreement 84
  2.3 Expansion: Peer-to-Peer payment 84
    2.3.1 Functionality 84
  2.4 Project goals 86
3. Software Development Methods 87
  3.1 Feature Driven Development 89
  3.2 The Model-View-Controller pattern 89
4. Used Technologies and Tools 91
  4.1 Apache, MySQL & PHP 91
  4.2 HTML, CSS, JavaScript (jQuery, AJAX) 91
  4.3 Zend framework 91
  4.4 PHPUnit 91
  4.5 Eclipse 92
  4.6 Selenium 92
  4.7 Github 92
  4.8 Browsers 92
5. Research 93
  5.1 Online rent payment systems 93
    5.1.1 RentPayment.com 93
    5.1.2 eRentPayment.com 94
    5.1.3 PayYourRent.com 95
  5.2 PayPal 95
    5.2.1 PayPal API 96
    5.2.2 Chained Payments 97
6. Timetable 99
References 101

1. Introduction

This project is commissioned by Roomplug LLC. Roomplug was in need of software development ninja's to help out on the rent payment system of their full service rental website. This is the preliminary report on the online rent payment system.

1.1 Roomplug

Roomplug is a start-up company founded in 2010 and based in San Clemente, California, USA. The company started with the mission of making online property management accessible to everyone. As landlords, both Rodney Nguyen (founder) and Sri Kanajan (co-founder) had a hard time managing their properties and their renters using existing services. They then decided to create Roomplug [1]. The online rental market already has property marketplaces, background and credit check services and rent payment systems available. Roomplug is unique because it puts all these services together into one system. From finding a room/roommate to signing the lease agreement to paying the rent. Everything is there.

2. Project Description

A part of the whole renting process is paying the rent. The common way to do this can be harsh, involving private bank account numbers and bank transfers. That's why Roomplug wants to have a peer-to-peer transaction system in which payments are done through the site.

While realising this system, state of the art web technologies will be used such as jQuery and AJAX to give the users an interactive experience.

2.1 Goal

The goal of the peer-to-peer payment feature is to make payments as insightful as possible to users and to enable renters and landlords to perform transactions seamlessly through the Roomplug website without having to share private banking details between the renter or the landlord.

To accomplish this goal, payments such as deposits, rent and late-payment fees have to be extracted from the contract between the renter and landlord. This way the landlord doesn't have to keep track of what payments he should receive and the renter has a clear overview on what to pay.

2.2 Current system

2.2.1 Users and advertisements

The current system is set up for two kinds of users: People who are looking for a room (renters) and people who have a room for rent (landlords). A user can also be both, since a user has access to all functionalities. For example, a user can add an advertisement stating he is looking for a room, but can also add an advertisement he has a room for rent, see Figure 1.

Figure 1: A user can have two different roles.

Users can create five advertisements for free but they can also pay for their advertisements to be an 'feature' ad. The ad will draw more attention to it because it is show in the featured ad listings on the right site of the website (except for the main page, it's show in the centre there).

2.2.2 Searching advertisements

As seen in the example, the current system already has the functionality of adding advertisements. The two different kinds of advertisements vary in their properties, but can both be found using the same search engine as seen in Figure 2. The search engine has several filter options and several advanced searching options.

Michael needs a room for rent in Boca Raton, FL, 33432
Hi, my name is Michael Richards, and I'm new to Roomplug.
38 Views 0 Comment 0 Plug
$0
5 months ago
Boca Raton, FL
Spacious 4 BR House to Share in Downtown Leesburg - Dog OK!
AVAILABLE IMMEDIATELY! I'm seeking a neat, quiet professional roommate with secure employment and a pleasant personality to share my home. It's a sp...
123 Views 0 Comment 0 Plug
$700
5 months ago
Leesburg, VA

Figure 2: Two different kinds of advertisements can be found with the same search engine.

2.2.3 Application and Lease agreement

After a user has found a suitable advertisement, he can apply for it, leaving behind personal information for review. The owner of the advertisement can then accept or reject the application. After accepting, a lease agreement can be created with various options. The final lease agreement is generated using the selected options and can be digitally signed by both parties.

2.3 Expansion: Peer-to-Peer payment

2.3.1 Functionality

This is the point where the new feature comes into play. The property owner defined the following information in the lease agreement:

- Rent;
- Rent due (on what interval is the rent payed);
- Deposit for the rent;
- Late fee's, if any:
    - Flat fee (a constant amount);
    - Per day fee (the late fee is increased by this amount each day);
- Number of days after payment date from which a payment is considered late;
- Return checks penalty.

The form is shown in Figure 3.

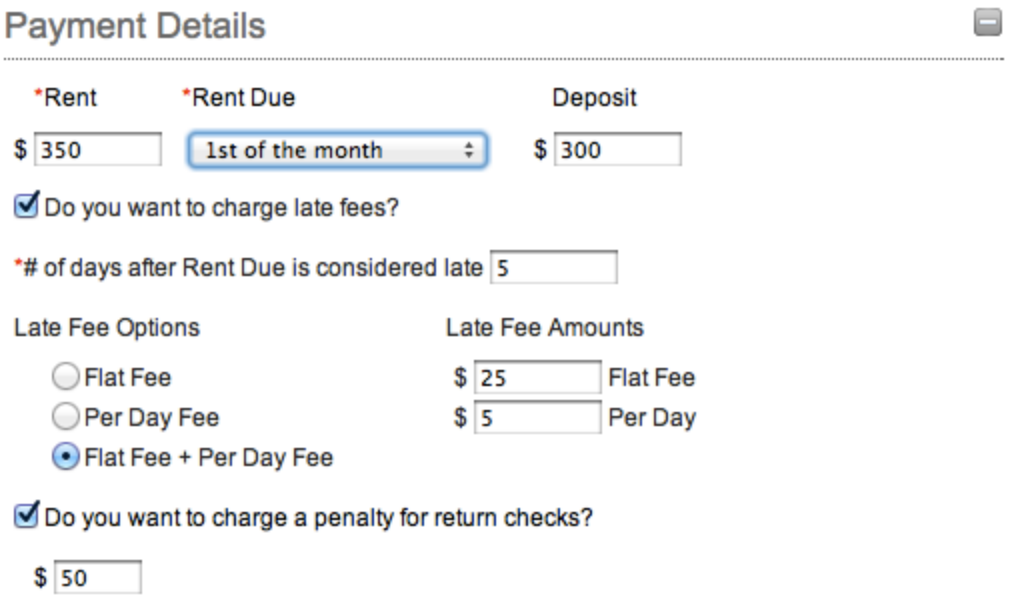

Figure 3: Payment details for a lease agreement.

The landlord can create invoices for a user and keep track of there status. Roomplug connects the invoices and payments to the correct users. This should all be done with the click of a button using the information from the lease agreement. It's should also be possible to make custom invoices.

On the other hand, it's also possible that a user has payed for something that should have been paid by the landlord. The renter can create a custom invoice for the landlord and keep track of it.

The eventual payment is done through PayPal which has an extensive API (later more on that). For every transaction a pre-defined fee will be payed to Roomplug. For now, PayPal is the only payment method available on Roomplug but there might be more payment methods added in the future so the payment system should be designed taking this in account.

2.4 Project goals

The project goals are shown in Table 1. The given format makes it reviewable by checking if the goals are met in the deliverables.

Action (short)	Action	Goal	Deliverables
Create requirements list.	Create a list of requirements for the online rent payment system.	The requirements should give a clear overview of the actions that users should be able to do.	Requirements list.
Create workflow diagrams.	Create workflow diagrams for both the landlords and the renters.	These workflows show exactly what steps a user needs to do in order to complete an action (as specified in the requirements).	Workflow diagrams.
Create global design.	Create global design of the payment system.	To get a clear model of the elements of the payment systems.	Design document.
Group actions/steps into features.	Group/divide the requirement actions and workflow steps into features.	This gives us a clear overview on the feature development milestones.	List of milestones.
Define core and additional features.	Define what milestones are core functionality and what features can be built on top of that.	Some functionality can be defined as a core feature. We need to define what is going to be the core and what is going to be built on top of that using the MoSCoW model.	List of core features and list of additional features.
	The following steps will be repeated for every feature:		
Design feature.	Design the details for a feature.	Creating a clear design on the functionality.	Design document.
Implement feature.	Implement the feature.	Building a working implementation of the feature design.	Code files.
Test feature.	Test the feature.	Making sure the feature works properly and future development doesn't break it.	Test Classes.

Table 1: Action list for reviewing the result.

3. Software Development Methods

In this chapter the development methods and patterns will be discussed that will be applied during the project. For the software building process the Feature Driven Development methodology is applied. The main pattern used in the project is the Model-View-Controller pattern.

3.1 Feature Driven Development

Feature Driven Development (FDD) is the most suitable method for the task at hand. This is because the required system can be easily split up in various different features. The basic features would be to make a payment to another user on the website. To integrate this in the whole Roomplug system, features will be added to improve the user experience by using existing data. FDD is an Agile form of developing [2].

First, an overall design is made with a high-level object model and notes as result. After this general overview, a features list is made, stating what features should be implemented. Those features can be grouped in related sets and subject areas. During this project, each set of features is considered a milestone, working from milestone to milestone, starting with the most fundamental ones.

For each feature (set) a more detailed design is made prior to implementation. Also, a set of test will be written to test the features. This can either be done after implementation, or parallel.

After all features are implemented, the project is completed. Failure to finish all features, will leave the system uncompleted, but it will still be able to work. This also illustrates the importance for the order in which features are implemented. A good way to prioritize them would be using the MoSCoW model, determining what features Must be implemented, Should be implemented, Could be implemented and Would be implemented.

3.2 The Model-View-Controller pattern

The Model-View-Controller (MVC) pattern is an architectural pattern used to seperate the functional part, the model, of the code from the views and its controllers [3]. This way it is easy to use written code in other parts of the system or even change the whole view without changing the code behind the interface. The Zend framework (more in Chapter 5.2) provides an easy way to set up the MVC pattern and to keep the structure organized.

Since this project has a web architecture, the MVC pattern is used somewhat differently as opposed to regular programming languages. The different responsibilities are now:

7. The View: Generates the HTML to display the page;
8. The Controller: Acts on POST transmissions, changing the Model, and sets variables for display in the view;
9. The Model: Manages the information in the database.

An overview of the MVC pattern in a web architecture is shown in Figure 4.

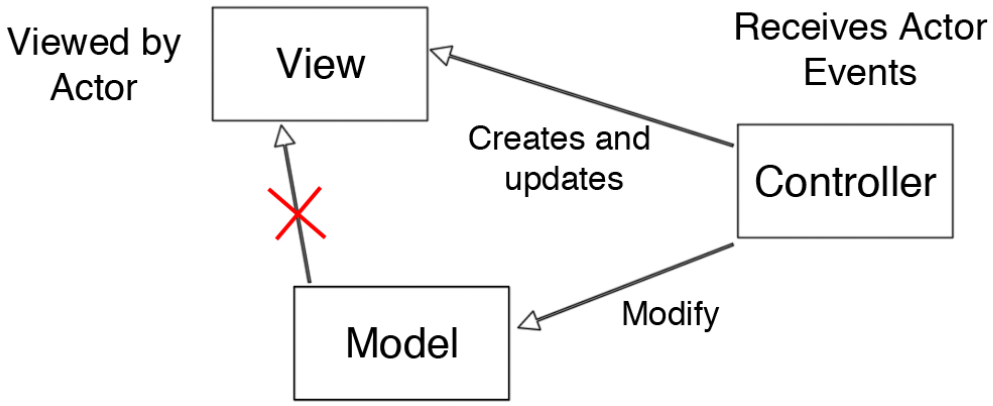

Figure 4: The MVC pattern in a web architecture. Note there is no notifaction for changes from the Model to the View.

Note the updates send from Controller to the View. These can be interpreted as the AJAX calls made from the client to the Controller, after which the Controller can send back information forcing the View to be updated.

4. Used Technologies and Tools

In this chapter, technologies and tools are shortly discussed that will be used in this project.

4.1 Apache, MySQL & PHP

The Roomplug code is written in the PHP script language and the data is stored in a MySQL database. The XAMPP package is used in order to develop and run the Roomplug website locally [4]. It contains the Apache web server with the PHP 5.2.17 extension on it and it also contains a MySQL server. Only little configuration tweaks are necessary to replicate the live environment of the Roomplug website.

4.2 HTML, CSS, JavaScript (jQuery, AJAX)

Since the portal of Roomplug is a website, HTML and CSS is used to display the actual pages. To give users an interactive experience, there are a lot of JavaScript applications which enables the developer to make changes to the HTML and CSS client side. jQuery is a JavaScript library, offering a lot out of the box JavaScript features, making it easier to use JavaScript. Also, AJAX calls can be made using jQuery. This allows the website to get new data from the server, or even make changes to the data, without a post back [5, 6].

4.3 Zend framework

The Roomplug code is built upon the Zend framework. This framework takes care of the more detailed part of programming so the programmer can focus on the bigger picture. The framework comes with a lot of standard reusable functionalities and it takes care of the communication with the database. The programmer only has to focus on what needs to be done instead of thinking of how it's done. On top of this, the Zend framework also supports the MVC design structure [7].

4.4 PHPUnit

Unit testing is done with PHPUnit which is incorporated in the Zend framework. It makes unit testing available to PHP developers in an easy to set up way. The tests will be written to test every feature before proceeding to the next feature [8].

4.5 Eclipse

Eclipse is an open source community, which focuses on building an open development platform [9]. There are numerous plugins for Eclipse, including plugins that enables developers to easily code in PHP (PHPeclipse [10]). Also, a plugin is available, EGit [11], that integrates the GitHub version control commands into the GUI, making it easier to communicate with the GitHub repository.

4.6 Selenium

Selenium is a suite of tools that provides a way to automate UI testing acros different browsers [12]. Using the IDE, which is a Firefox plugin, user actions can be recorded and assertions can be added, which can then be played back on different browsers using Selenium Remote Control [13]. This way a test suite can be easily created and used on multiple browsers. Also, after generating the tests, they can be exported to PHPUnit so they can be tweaked further.

For this project, UI testing will be done manually at first since the overhead of creating the test suite is to big. The need for a test suite will be reviewed later on in the project.

4.7 Github

Git is a free and open source version control system and GitHub is a web-based cloud hosting service for it. Git and GitHub boost collaboration in (distributed) collaborative software development. Roomplug has its repository hosted here [14, 15].

4.8 Browsers

The Roomplug website will be visited by people using different browsers. The source code should follow the web standard guidelines to ensure that all browsers will display the website as correct as possible. Unfortunately, every browser will have it's own way of rendering certain parts of a website. Therefore, it's important that the website is tested on different browsers. These test should a least include the latest (few) versions of popular browsers like Google Chrome, Mozilla Firefox and Microsoft Internet Explorer. These browsers are being used by more than 90% of the internet users at the moment [16].

5. Research

5.1 Online rent payment systems

Roomplug has already done research on competing sites with similar functionality or parts of it. Like stated before, there are already some online rent payment systems available. Looking into those existing systems gives an idea of what should be included in the system. Unfortunately, none of the online payment systems websites provide a usable demo but the websites do give a general overview over all the functionality.

5.1.1 RentPayment.com

RentPayment.com is the largest online rent payment company in the USA. They're part of a larger group of online payment systems and have received large amounts of funding. This results in a system with lots of possibilities to make payments: rentpayment.com website, phone, text messaging, fax and email. There's even an iPhone app. The website and iPhone app offer the possibility to set-up auto payments so there's no need to worry about paying in time. The payments can be done by debit, credit card or e-check.

RentPayments.com also offers functionality to integrate payments and property with the accounting and property management software used by the landlords and renters. This makes it possible for new users to start using RentPayment.com without any problems [17].

Figure 5: The RentPayment.com iPhone app.

5.1.2 eRentPayment.com

eRentPayment.com is another big online rent payment system. The offer single and recurring transctions which can be don either through the website or by phone. It's possible for renters to authorize the payments to the landlords without using the website. They only need to fill out a form which authorizes a recurring payment.

Instead of worrying about payment methods themselves, eRentPayment.com uses the Automated Clearing House (ACH) network. ACH is a large financial transaction network in the USA taking care of debit and credit checking and processing transactions by various means including payment by credit card [18, 19, 20].

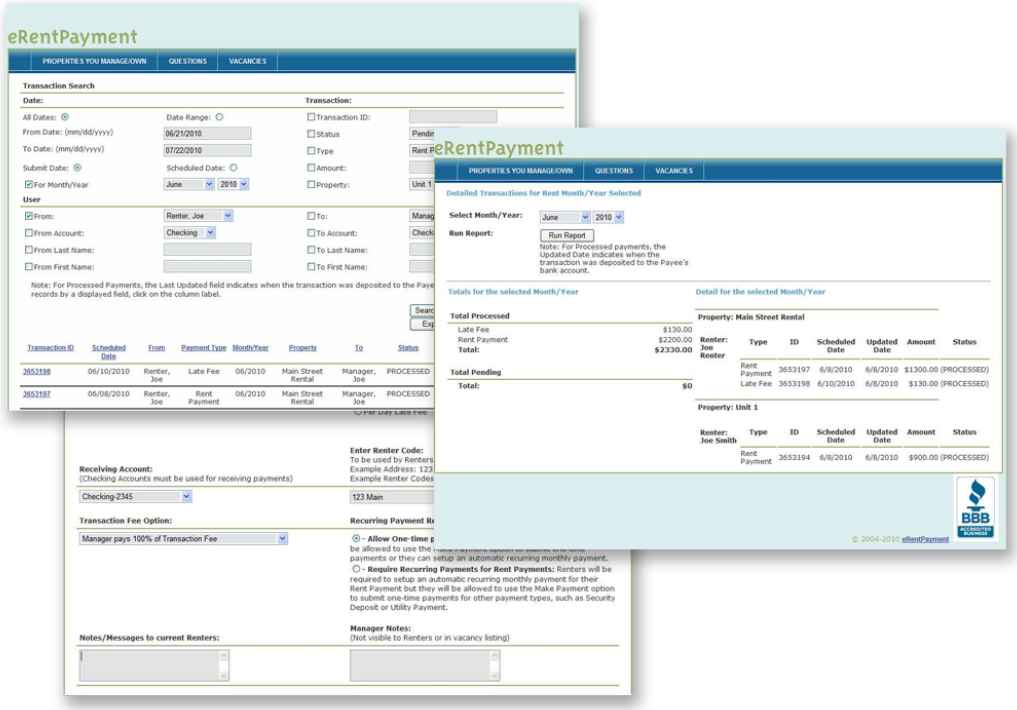

Figure 6: The eRentPayment.com manager interface.

5.1.3 PayYourRent.com

PayYourRent.com offers an elaborate manager for both landlords and renters. Renters can only pay the rent through the website with one-time payments or they can set-up automatic recurring payments. The manager also supports reporting maintenance issues to the landlords and landlords can post announcements to renters. It's also possible to take care of property insurance through the website.

Like eRentPayment.com, PayYourRent.com supports credit cards and e-checks which are processed by the ACH network [20, 21, 22].

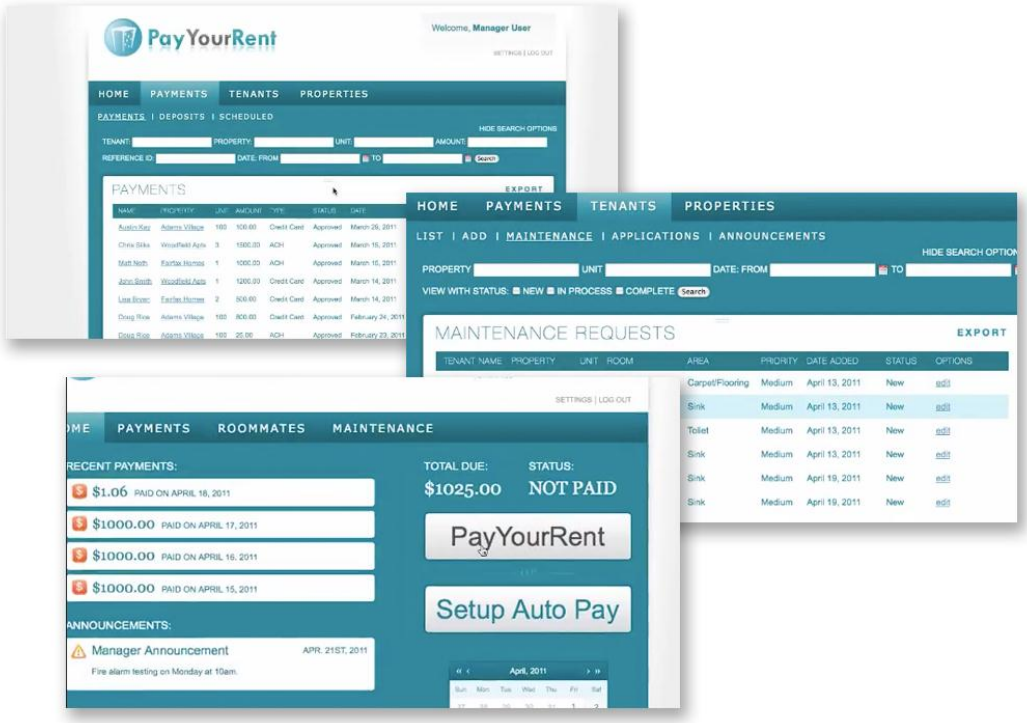

Figure 7: The eRentPayment.com manager interface.

5.2 PayPal

Roomplug uses PayPal as it's main provider for financial transactions. It's already in use for the payed services of the website, like the featured ads. PayPal will also be the main provider used for the rent payment system although the system will be designed so that other payment providers are easily added.

5.2.1 PayPal API

PayPal has an extensive API and it also provides sandbox possibilities for testing purposes [23, 24]. This makes it easy to develop and test the payment functionality on the Roomplug systems. Also, a SDK is provided, which makes it easier to use the available API operations [25].

An overview of the basic Adaptive API [26] operations are shown in Figure 8.

API Operations Related to Payments

API Operation	Description
Pay	Transfers funds from a sender's PayPal account to one or more receivers' PayPal accounts (up to 6 receivers)
PaymentDetails	Obtains information about a payment created with the Pay API operation
ExecutePayment	Executes a payment
GetPaymentOptions	Obtain the settings specified with the SetPaymentOptions API operation
SetPaymentOptions	Sets payment options

API Operations Related to Preapprovals

API Operation	Description
Preapproval	Sets up preapprovals, which is an approval to make future payments on the sender's behalf
PreapprovalDetails	Obtains information about a preapproval
CancelPreapproval	Cancels a preapproval

Other API Operations

API Operation	Description
Refund	Refunds all or part of a payment
ConvertCurrency	Obtains the current foreign exchange (FX) rate for a specific amount and currency
GetFundingPlans	Determines the funding sources that are available for a specified payment
GetShippingAddresses	Obtains the selected shipping address

Figure 8: An overview of the API operations [27].

5.2.2 Chained Payments

PayPal provides so-called chained payments, which allows Roomplug to take a cut of the payed amount and forwarding the remainder to the receiver. This is shown in Figure 9.

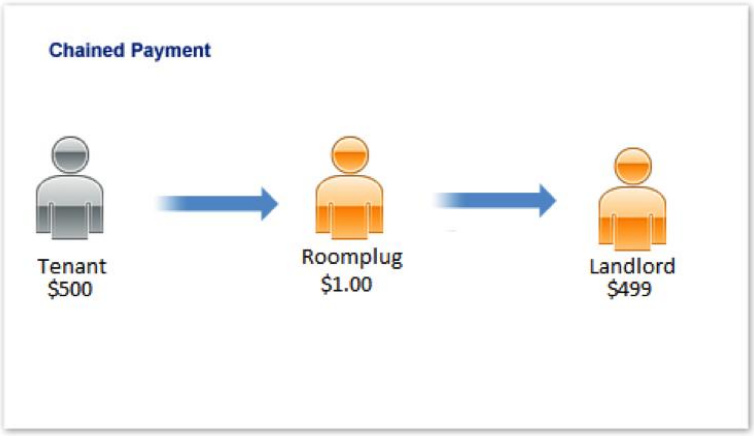

Figure 9: Chained payment using PayPal. Note that PayPal fees are not included.

In order to use these advanced features of PayPal, the application must meet the following requirements:

5. A premier or business PayPal account is needed;
6. Permissions to the Adaptive API need to be given in the account settings;
7. The implementation needs to be verified, guidelines given on [28, 29, 30].

6. Timetable

The following overview gives a written description on the plan:
Weeks 1-3:

13. Get to know the development environment (Zend framework, jQuery & AJAX, PHPUnit, Github);
14. Research existing Peer-to-Peer payment systems;
15. Review PayPal API's;
16. **Deliverables**: Preliminary report.

Weeks 4-5:

18. Make overall design, high-level object model and feature list;
19. Group features in sets;
20. Determine the order of implementation using the MoSCoW model;
21. Determine actual milestones and deadlines for each.

Weeks 6-8:

19. Make feature level design and implementation for core features;
20. Write tests for core features;
21. **Deliverables**: Code for review by SIG.

Weeks 9-12:

22. Make feature level design and implementation for extending features;
23. Write tests for extending features;
21. **Deliverables**: Final report.

November:

- Present the work at DUT.
- **Deliverables**: Presentation.

Since the various phases overlap each other, a graphical overview is given in Figure 10.

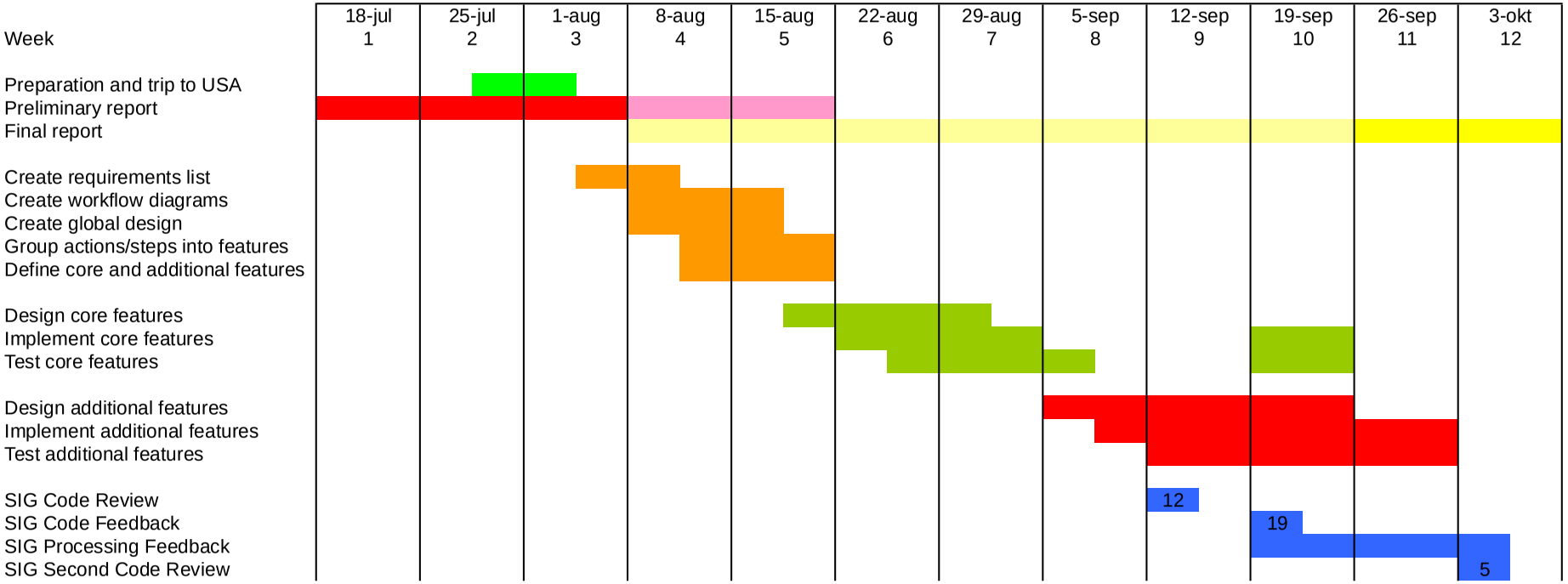

Figure 10: A graphical view on the timetable.

References

[1] “About Roomplug”, Roomplug. Visited on July 27, 2011. <http://www.roomplug.com/about>.

[2] “Feature Driven Development (FDD) and Agile Modeling”, Ambler, S.W. Visited on July 29, 2011 <http://www.agilemodeling.com/essays/fdd.htm>.

[3] Lethbridge, T.C. and Laganière, R., Object-Oriented Software Engineering, second edition, McGraw-Hill Education, UK, 2005, pp. 355-357.

[4] “XAMPP”, Apache Friends. Visited on July 29, 2011. <http://www.apachefriends.org/en/xampp.html>.

[5] “jQuery”, The jQuery Project. Visited on August 1, 2011. <http://jquery.com/>.

[6] “jQuery.AJAX”, The jQuery Project. Visited on August 1, 2011. <http://api.jquery.com/jQuery.ajax/>.

[7] “Programmer’s Reference Guide”, Zend technologies Ltd. Visited on July 29, 2011. <http://framework.zend.com/manual/en/learning.quickstart.intro.html>.

[8] “Unit Testing with the Zend Framework with Zend_Test and PHPUnit”, Zend technologies Ltd. Visited on July 29, 2011. <http://www.zendcasts.com/unit-testing-with-the-zend-framework-with-zend_test-and-phpunit/2009/06/>.

[9] “Eclipse”, The Eclipse Foundation.Visited on August 1, 2011. <http://www.eclipse.org/>.

[10] “PHPeclipse”, PHPeclipse. Visited on August 1, 2011. <http://www.phpeclipse.com/>.

[11] “EGit”, The Eclipse Foundation. Visited on August 1, 2011. <http://eclipse.org/egit/>.

[12] “SeleniumHQ”, Selenium. Visited on August 1, 2011. <http://seleniumhq.org/>.

[13] “Selenium Remote Control”, Selenium. Visited on August 1, 2011. <http://seleniumhq.org/projects/remote-control/>.

[14] “About GIT”, GIT. Visited on August 1, 2011. <http://git-scm.com/about>.

[15] “About GitHub”, GitHub Inc. Visited on August 1, 2011. <https://github.com/about>.

[16] “Browser Statistics”, w3schools. Visited on August 1, 2011. <http://www.w3schools.com/browsers/browsers_stats.asp>.

[17] “RentPayment products”, YapStone Inc. Visited on August 1, 2011. <http://www.rentpayment.com/products/index.html>.

[18] “Online Rent Payment”, eRentPayment. Visited on August 1, 2011. <https://www.erentpayment.com/renters.aspx>.

[19] “Four Payment Options to offer your Renters”, eRentPayment. Visited on August 1, 2011. <https://www.erentpayment.com/managers.aspx>.

[20] “ACH processing”, ACH Payments. Visited on August 1, 2011. <http://www.ach-payments.com/ach-processing.htm>.

[21] “Payments”, PayYourRent. Visited on August 1, 2011. <http://www.payyourrent.com/index.php?option=com_content&view=article&id=34&Itemid=48>.

[22] “FAQ”, PayYourRent. Visited on August 1, 2011. <http://www.payyourrent.com/index.php?option=com_content&view=article&id=40&Itemid=83>.

[23] “API Integration Summary”, PayPal X. Visited on July 29, 2011. <https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/howto_api_overview>.

[24] “PayPal Sandbox”, PayPal X. Visited on July 29, 2011. <https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/howto_testing_sandbox>.

[25] “PayPal SDKs and Downloads”, PayPal X. Visited on August 1, 2011. <https://www.x.com/community/ppx/sdks>.

[26] “Adaptive Payments API”, PayPal X. Visited on August 1, 2011. <https://www.x.com/community/ppx/adaptive_payments>.

[27] “Getting started”, PayPal X. Visited on August 1, 2011. <https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/e_howto_api_APGettingStarted>.

[28] “Adaptive Apps 101”, PayPal X. Visited on August 1, 2011. <https://www.x.com/community/ppx/apps101>

[29] “Building Apps Using Adaptive APIs”, PayPal X. Visited on August 1, 2011. <https://www.x.com/community/ppx/apps101/building-apps>.

[30] “What You Need to Get Your Adaptive APIs App Live”, PayPal X. Visited on August 1, 2011. <https://www.x.com/community/ppx/apps101/go-live>.

Appendix D: Manuals

This appendix contains manuals for both developers, administrators and regular users.

D.1 User manual

Making a custom payment

A Roomplug user must to navigate to its hub and click the 'Payments' tab to start making a payment. The user will then see the payment overview page show in Figure D.1.1.

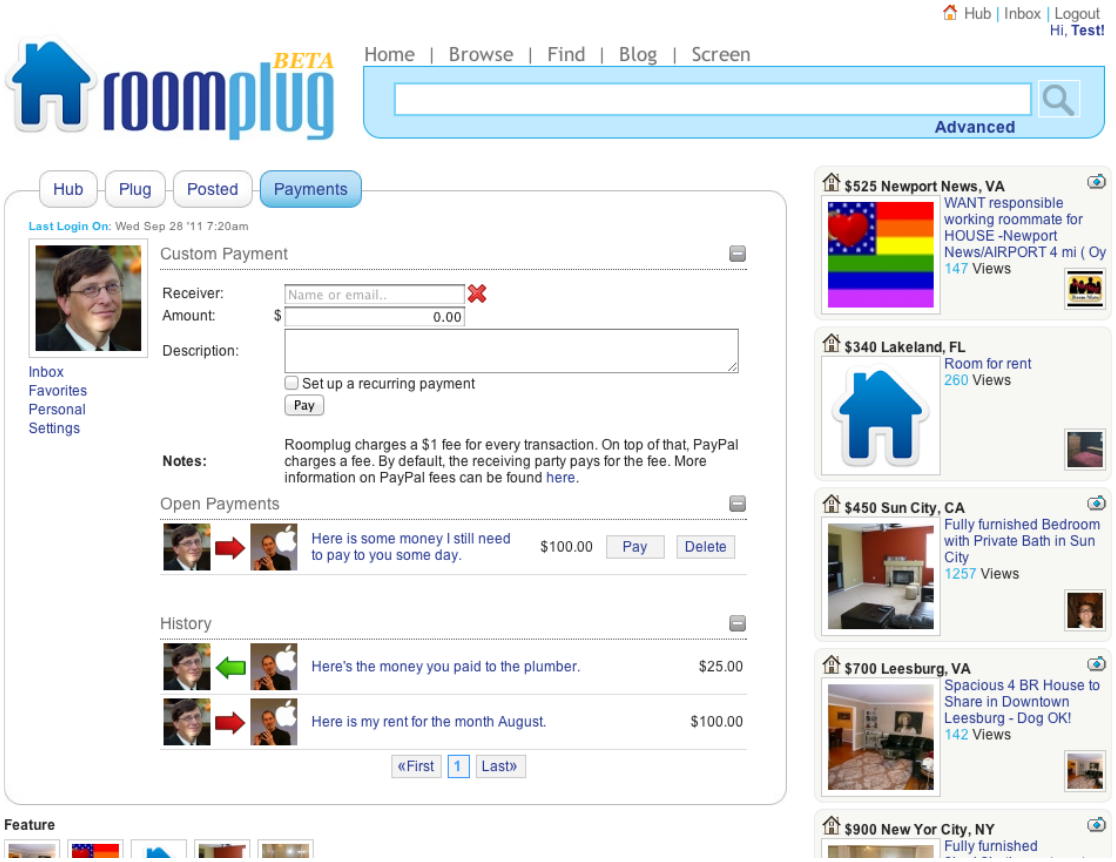

Figure D.1.1: The overview page

The user can enter the payment data on this page in the form under the header ‘Custom Payment’. The user selects the Roomplug user that receives the payment, by typing (part of) its name, or the full email address of the user to send money to. This is illustrated in Figure D.1.2.

Figure D.1.2: Selecting the receiving user

After a receiving user is selected, the amount and description of the payment can be entered, as illustrated in Figure D.1.3. After all this is done, the user clicks the 'Pay' button. This will take the user to the summary page.

Figure D.1.3: The Custom Payment form with all data filled in

The summary page, shown in Figure D.1.4, displays the payment details. The user can check the data, prior to picking the payment method. In this case, only PayPal is available. The user has to click the PayPal 'Pay Now' button to continue with the payment.

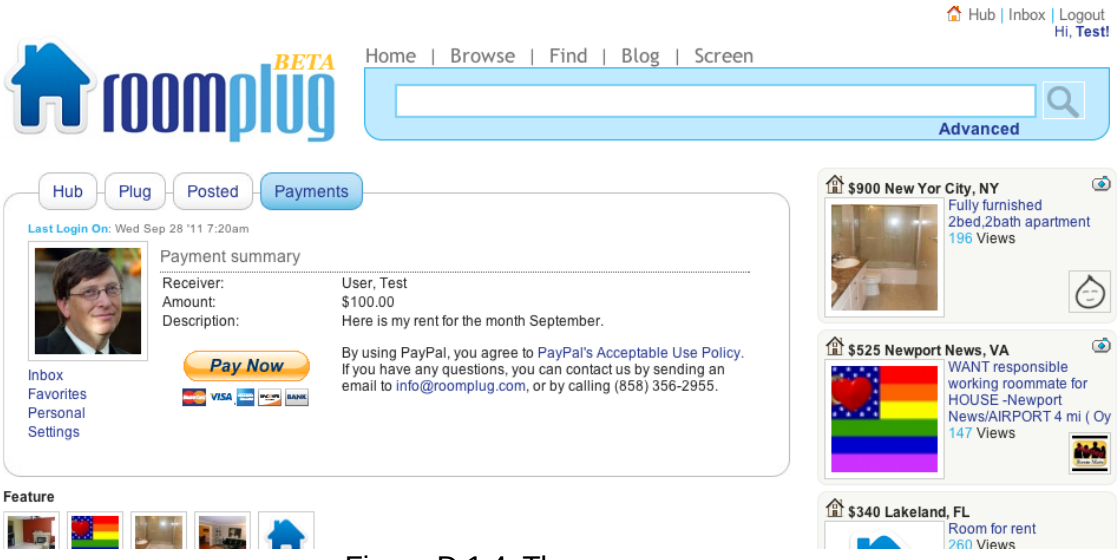

Figure D.1.4: The summary page

The user will then be taken into the PayPal payment system to execute the actual payment, as seen in Figure D.1.5.

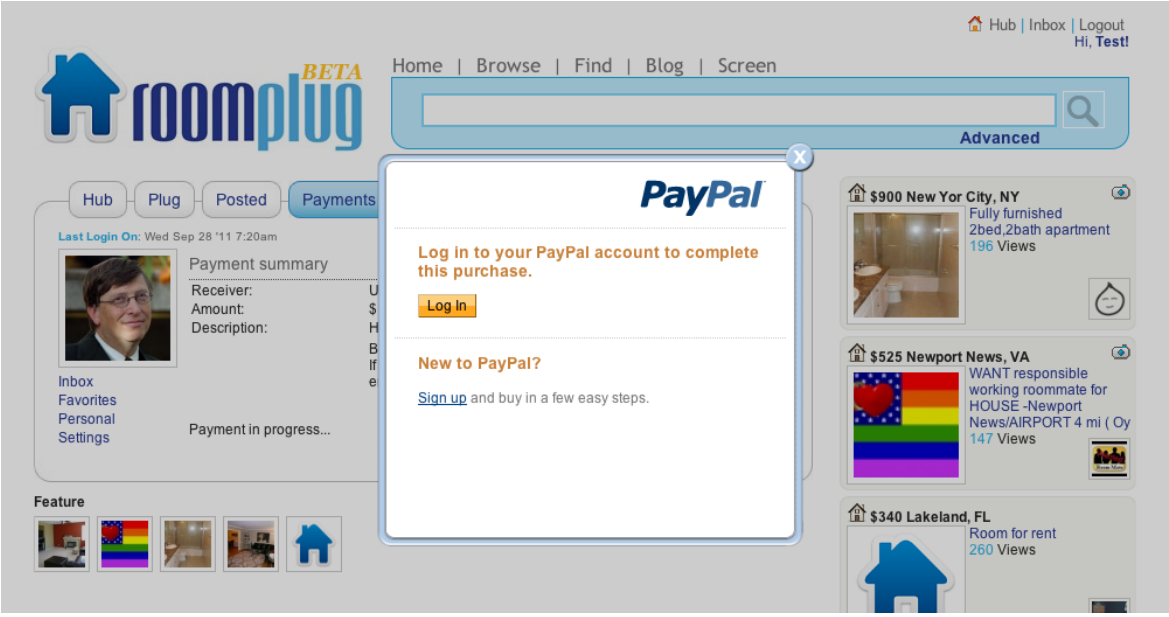

Figure D.1.5: The PayPal overlay

After the user finishes the payment in the PayPal system, the user will see the success message and a link back to the overview page, illustrated in Figure D.1.6.

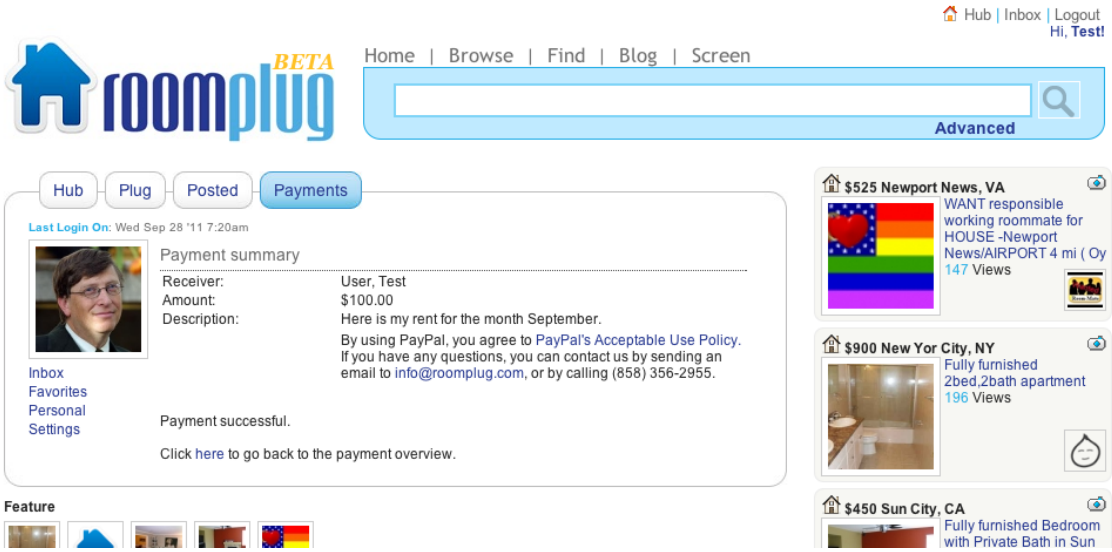
Hub | Inbox | Logout
Hi, Test!
roomplug BETA
Home | Browse | Find | Blog | Screen
Advanced
Hub
Plug
Posted
Payments
Last Login On: Wed Sep 28 '11 7:20am
Payment summary
Receiver: User, Test
Amount: $100.00
Description: Here is my rent for the month September.
By using PayPal, you agree to PayPal's Acceptable Use Policy.
If you have any questions, you can contact us by sending an email to info@roomplug.com, or by calling (858) 356-2955.
Inbox
Favorites
Personal
Settings
Payment successful.
Click here to go back to the payment overview.
Feature
$525 Newport News, VA
WANT responsible working roommate for HOUSE -Newport News/AIRPORT 4 mi (Oy
147 Views
$900 New Yor City, NY
Fully furnished 2bed,2bath apartment
196 Views
$450 Sun City, CA
Fully furnished Bedroom with Private Bath in Sun

Figure D.1.6: The success page

The overview page now shows the newly paid payment in the payment history, shown in Figure D.1.7.

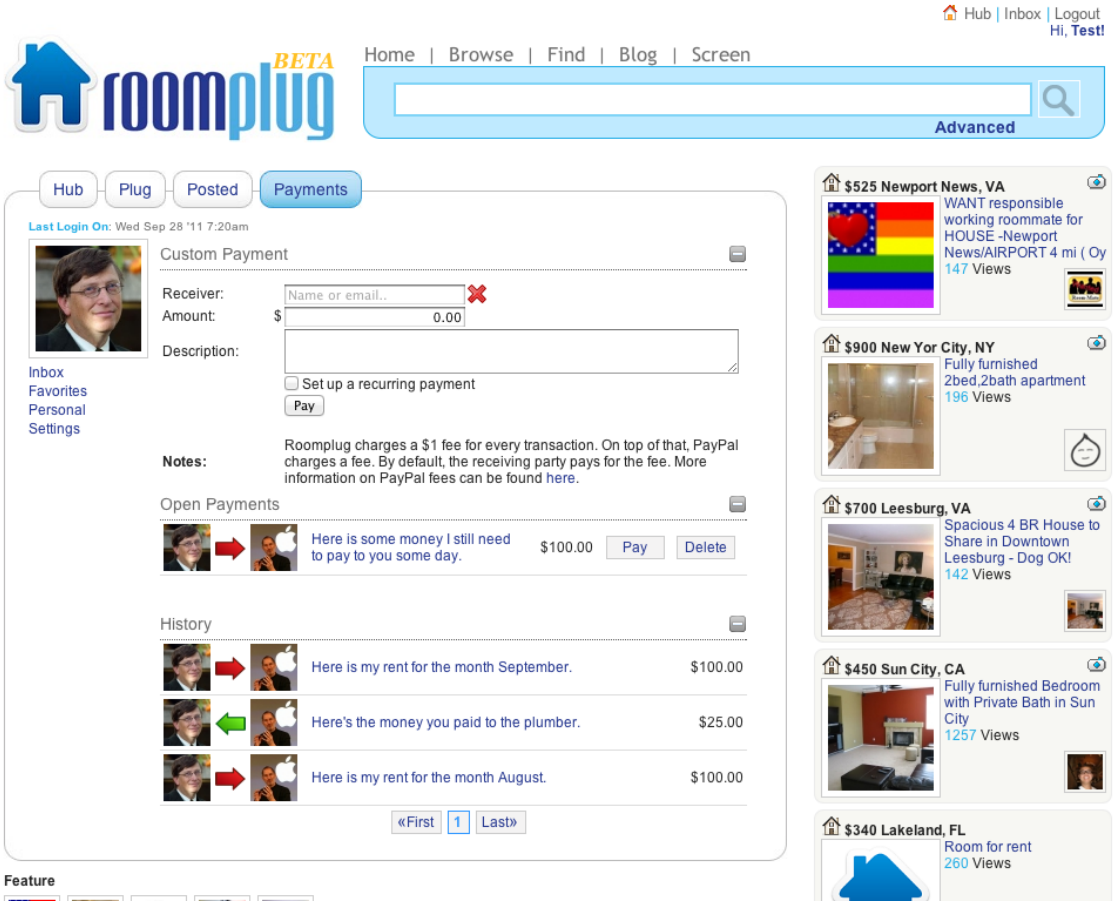

Figure D.1.7: The overview page showing the processed payment

Making a guest payment

Making a guest payment works pretty much the same as making a custom payment. The only difference is that a user is not logged in and it has to fill in an extra form to initiate the payment. This form includes the first name, last name, and email address of the guest, shown in Figure D.1.8. The guest payment form can be found here: http://www.roomplug.com/pay/guest.

Figure D.1.8: The guest payment page

After filling in all the data in this form, the user will be taken to the summary page. From here on, everything works the same as in the creation of a custom payment.

Set up a recurring payment

To set up a recurring payment, the user has to have a Roomplug account and log in into the system. After navigating to the 'Payments' tab on the users hub, the user can start entering the payment data in the same way as they would do while creating a custom payment. To make the payment recurring, the user has to check the 'Set up a recurring payment' checkbox. Once checked, extra options regarding the frequency of payments to occur and the period wherein they occur are shown. This can be seen in Figure D.1.9. The user has to fill in the extra options and click 'Pay'. This will take the user to the summary page.

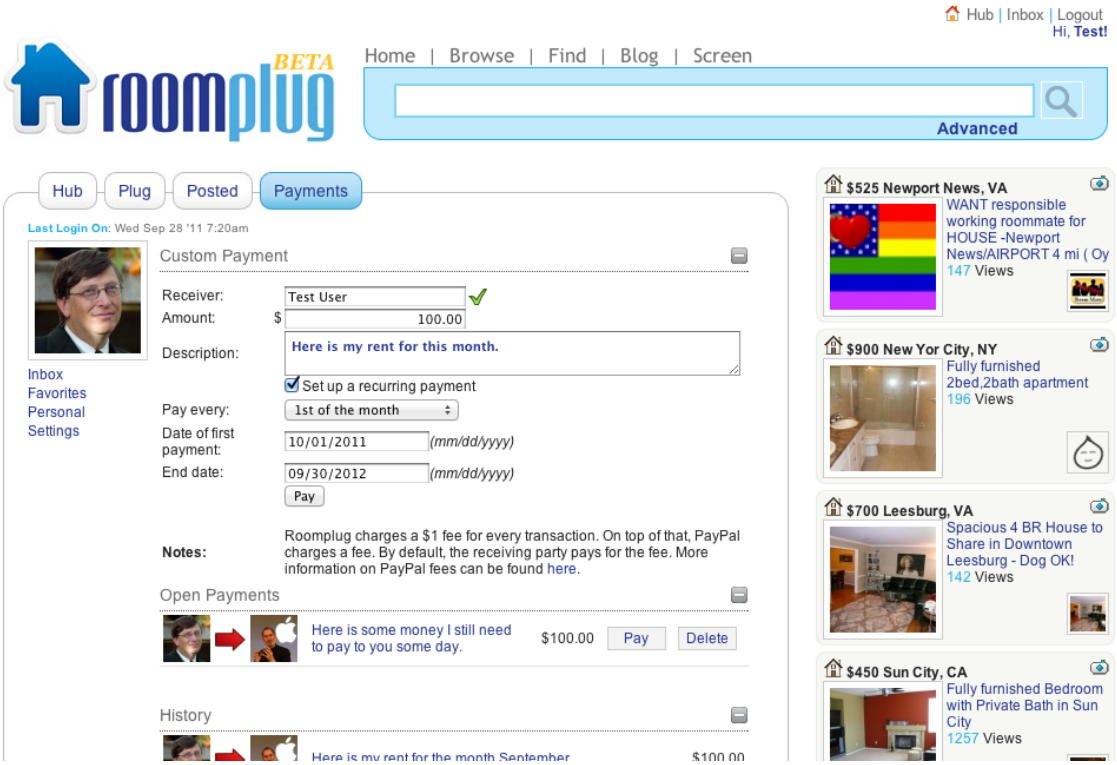

Figure D.1.9: The overview page showing the recurring payment fields of the Custom Payment form

The summary page looks somewhat like the summary page of the custom payment. The only difference is that some extra details regarding the recurring payment will be shown, as can be seen in Figure D.1.10. Just below the details, the user can choose their payment method. This will take them into the PayPal system.

Figure D.1.10: The summary page showing the recurring payment details

As seen in Figure D.1.11, this screen looks very different from the custom payment, because the user does not actually execute a payment. The user pre-approves Roomplug to automatically execute the payments for them in the future.

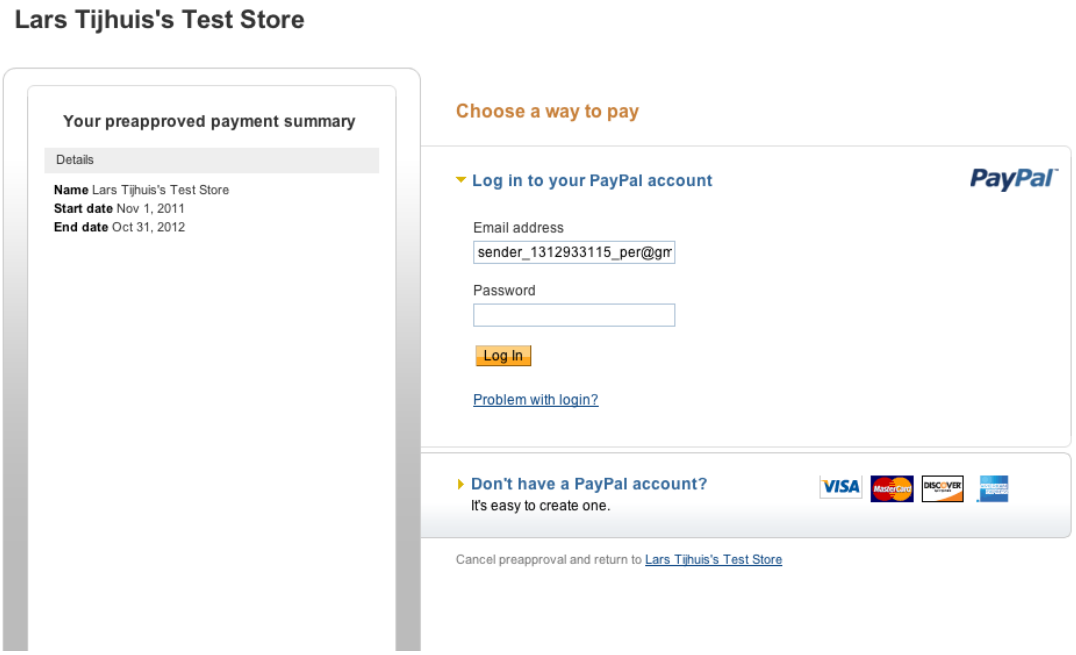

Figure D.1.11: The PayPal screen for the pre-approval of a recurring payment

Once the user finishes the pre-approval process in the PayPal system, they will be taken back to the Roomplug system. The user will see the success page telling them that the pre-approval was successful, as illustrated in Figure D.1.12.

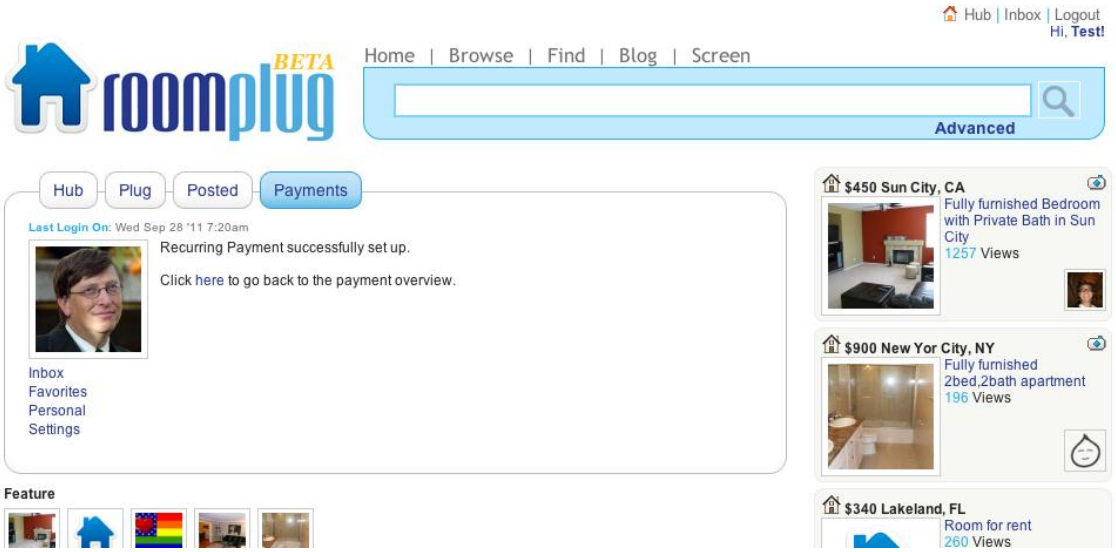

Figure D.1.12: The success page of the pre-approval of a recurring payment

After navigating back to the overview page, the user will see the recurring payment being active. This can be seen in Figure D.1.13.

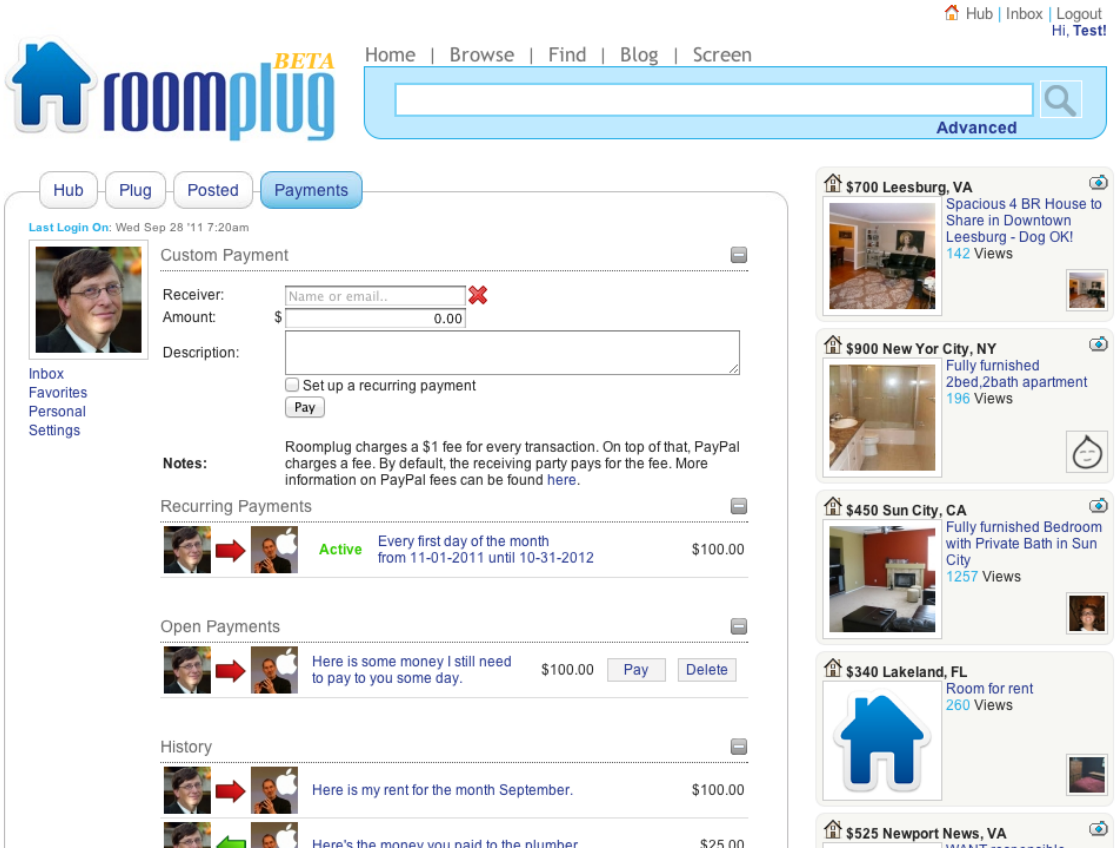

Figure D.1.13: The overview page showing the approved and active recurring payment

Manage payments

The overview page is the main entry page to the payment system and is shown in Figure D.1.14. This page is the starting point for all user actions including managing their payments. At first there is the custom payment header. The contents of this header has already been discussed in the section about creating a payment. This section will discuss all the managing actions of the other elements on this page.

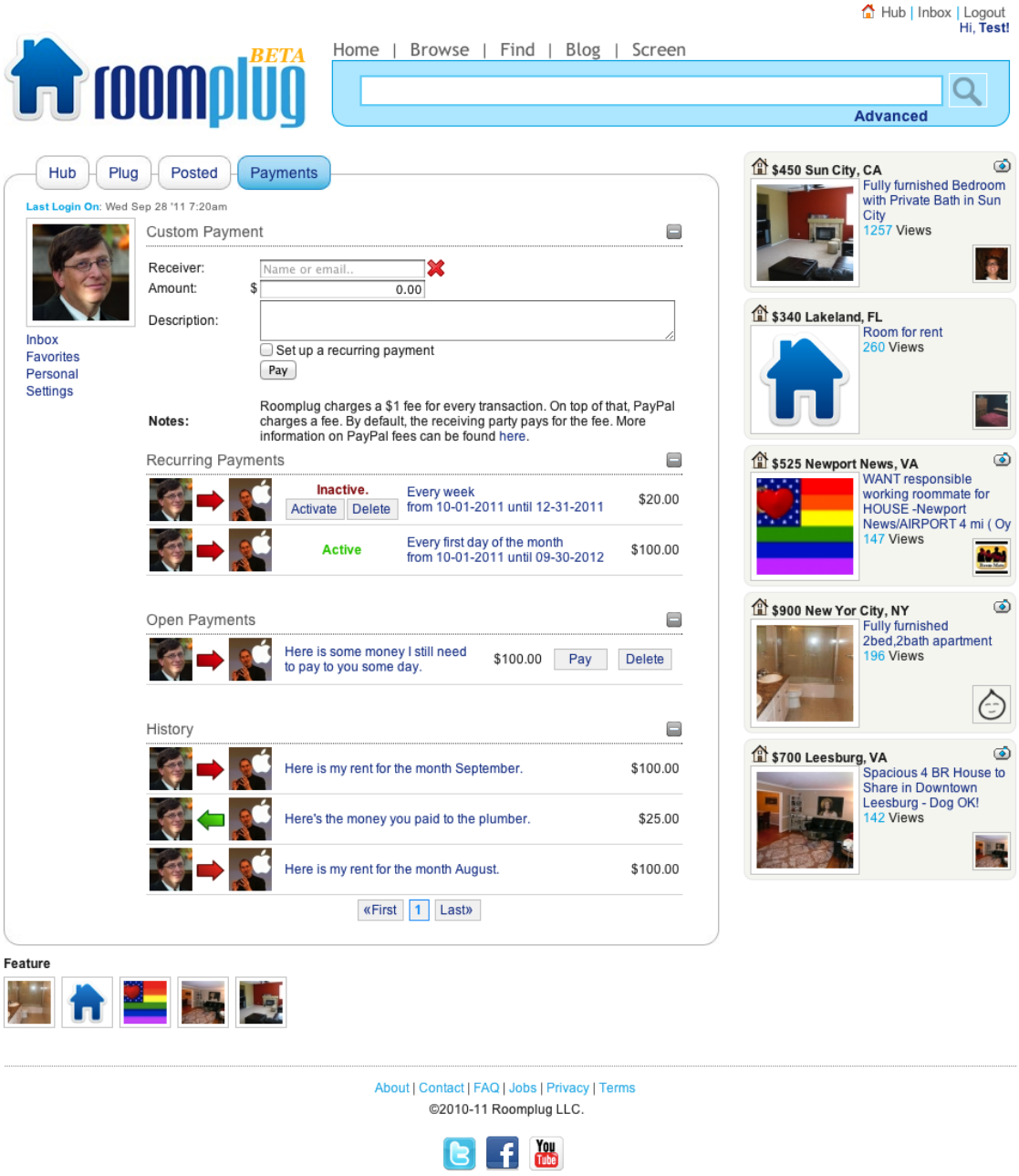

Figure D.1.14: The overview page showing all kinds of payments

The first header after ‘Custom Payments’ is called ‘Recurring Payments’. This header will only show if there actually are active incoming or an outgoing recurring payment. If the recurring payment is not activated, the user gets the option to activate it or to delete it. The meaning of the pictures with the arrow in between them will be discussed in the end of this section since all the headers have this.

Then there is the ‘Open Payments’ header. This shows all open payments that have not been paid yet. The user gets the option to pay or delete the payment. If the user decides to pay it, they will be taken to the summary page to select their payment method.

Last but not least, there is the ‘History’ header. This will show all the incoming or outgoing payments that have been paid. Only the last ten payments will be shown, since the number of payments under this header grows over time. The user can use the page controls to view older payments.

And then there are some elements that all headers have in common. Each payment row shows the user images that participated in the transaction. In between them is either a red or a green arrow, which shows the direction of the money flow and thus what user is the paying user and what user is the receiving user. Red arrows are always pointing to the right and they indicate that the payment is an outgoing transaction. Green arrows always point to the left and they indicate an incoming transaction.

All the describing text on the payments is clickable. Clicking on it will take the user to the details page. The text itself will become underlined and the mouse cursor will change into a pointer once a user hovers over the text. This signals the user that the text is clickable.

Figure D.1.15: The details page of a recurring payment

The details page is fairly simple since it only shows the (recurring) payment details, as illustrated in Figure D.1.15. The page will also show buttons to do actions like ‘pay’ or ‘delete’ on this payment if those actions are possible.

Set up account

In order to receive payments, Roomplug users have to set up their PayPal account in their Roomplug profile. From the hub, the users can click the ‘personal’ link under their profile picture. Then they have to click on the ‘My PayPal Account’ header. This will reveal the PayPal account verification form, as shown in Figure D.1.16. Once the data that has been entered is verified, the user is all set to receive payments.

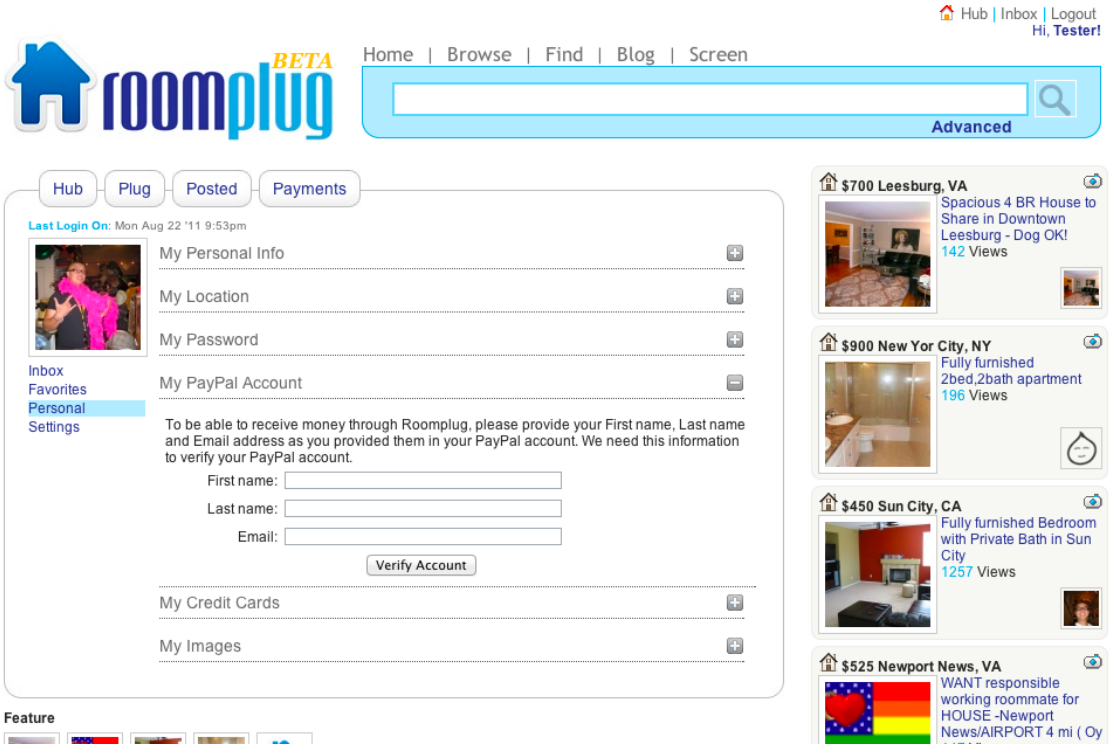
Hub | Inbox | Logout
Hi, Tester!
roomplug BETA
Home | Browse | Find | Blog | Screen
Advanced
Hub
Plug
Posted
Payments
Last Login On: Mon Aug 22 '11 9:53pm
My Personal Info
My Location
My Password
My PayPal Account
Inbox
Favorites
Personal
Settings
To be able to receive money through Roomplug, please provide your First name, Last name and Email address as you provided them in your PayPal account. We need this information to verify your PayPal account.
First name:
Last name:
Email:
Verify Account
My Credit Cards
My Images
Feature
$700 Leesburg, VA
Spacious 4 BR House to Share in Downtown Leesburg - Dog OK!
142 Views
$900 New Yor City, NY
Fully furnished 2bed,2bath apartment
196 Views
$450 Sun City, CA
Fully furnished Bedroom with Private Bath in Sun City
1257 Views
$525 Newport News, VA
WANT responsible working roommate for HOUSE -Newport News/AIRPORT 4 mi (Oy

Figure D.1.16: The profile page showing the PayPal account info form

IPN and payment troubleshooting

It might happen that a payment is executed in the PayPal system, but the IPN calls did not get through to the Roomplug system. PayPal will try to resend the IPN call for four days in intervals that increase in size. Although it is unlikely, it might be possible that a payment will not be verified at all.

If a user complains about a situation like this, Roomplug can log into PayPal with their business account and manually check the payment using the PayPal token as a unique identifier. The payments always go through the PayPal application of Roomplug, so it is possible to check the users claim. When the user is right, Roomplug can request a new IPN call in the PayPal system or manually update the data in the database.

It can also happen that an IPN call did get through but PayPal does not verify it. This can be the result of someone faking a call or a failure on PayPals side. The Roomplug system logs every call and every step of the IPN listener. Roomplug can use this log in the database to check what went wrong and take actions accordingly.

D.2 Maintenance manual

Adding a new payment method

Two different parts can be identified that need changes when adding a new payment system: the model and the use of the adapter to execute payments.

The *PaymentConnector* and the other *connector* classes are created to make expanding the model with a new payment method easier. The only thing that needs to be done is extending the *connector* classes with classes containing specific information for that payment method. Very little work needs to be done in the existing classes.

For example, you want to add a new system called 'MakeMeRich'. MakeMeRich provides an API with which you can send identification and payment information without referring the user to their website. For this, you need 2 properties called 'foo' and 'bar' that are unique for a payment. In order to achieve this, the following actions need to be taken:

1. Add a database table containing a foreign key to a payment in the *rp_payments* table;
2. Create the corresponding class *MakemerichPayments*, extending *Zend_Db_Table_Abstract*. Implement retrieval methods in this class specific to this payment type. Also, ensure that the properties *$_name*, *$_primary* and *$_rowClass* are set. An example can be found in the *PaypalPayments* class;
3. Extend the *PaymentConnector* with a new class called *MakemerichPayment*, containing getters and setters for the *foo* and *bar* properties. Those can be similair to the getters and setters in *PaypalPayment*.

Now your objects are ready.

In order to keep the execution of payments generic, an abstract class *PaymentAdapter* is created. For the MakeMeRich method, we add a new adapter called *MakemerichAdapter*, extending the *PaymentAdapter*. In this adapter, implement the functions *executePayments*, *setUpRecurringPayment* and *paymentsAreValid*. The actions in those functions taken are specific to the API and flow of the MakeMeRich system. For example, with PayPal those functions do the following:

- *executePayments*: Retrieve a token for the given payment from PayPal;
- *setUpRecurringPayment*: Retrieve a pre-approval token for the recurring payment from PayPal;
- *paymentsAreValid:* Checks PayPal specific constraints for a payment.

In the PayPal flow, the *executePayments* and *setUpRecurringPayment* functions are used in AJAX calls from the summary page. Adding a new payment method would involve creating similar calls. A new controller should be created with actions specific to this new payment method.

For the *paymentsAreValid* function,there is one change that needs to be done when adding a the MakeMeRich payment method. Currently, the PayPal specific checks are done right after the general input checks and, on error, returned with them. When adding the MakeMeRich payment method this is not the wanted behaviour. PayPal may restrict a certain condition, but MakeMeRich may not. If MakeMeRich would allow the payment to be made and PayPal does not, the wanted behaviour is to show the summary page and disable the option to pay with PayPal with the reason why it is disabled. Users can then still choose to use MakeMeRich. The place to do the checks can remain the same, the way the response is handled should change.

When the new payment method supports recurring payments, the recurring payment cronjob needs to be updated as well. The *recurringpaymentAction* is set up to create occurrences of recurring payments when needed, functionality to do this for the new payment method must be added.

Note:
Pay attention that when changing or adding a method to the *Payment* class, a corresponding method using this method needs to be added to the *PaymentConnector* class (or a similar class and connector). This is a weak point of the approach chosen, but this should only occur when a fundamental change is made to a Payment.

Cronjob

The Roomplug system must take care of executing payments, for the recurring payments in the system. The recurring payment script that does this can be found in the *CronjobController*. It runs twice a day using a cronjob, which is configured on the webserver. It runs twice for redundancy reasons, the second run will backup the first one in case something goes wrong.

The cronjob is currently set to run at 11:55 AM and 11:55 PM and it executes the following command:

```
curl http://www.roomplug.com/cronjob/recurringpayment?secret_password=[secret password]
```

The '[secret password]' in this command should be replaced with the actual password, which is set in the *init* function of the *CronjobController*. The password is set to prevent random people from starting the recurring payment script by navigating to the address. It does not hurt if only one user does this, but the server will be overloaded if hundreds of users do. The script is constructed as a webpage, and therefore accessible to the public, because its the easiest way to bootstrap Zend Framework and load all the models.

D.3 Test manual

How it is structured

Almost every source file in the payment system, has a *mirrored* test file in the test suite. The ones that do not, are tested by using related classes. Such an example is the *Payments* class, which aligns with the *Payment* class. This *mirroring* of files can be seen in the file structure of the test suite.

The *phpunit.xml* file defines what file is used to bootstrap the Zend Application, where the test suite can be found, and what source files should (or should not) be analyzed on code coverage. An example of the contents can be found in Code sample D.3.1.

```
<phpunit bootstrap="./application/bootstrap.php">
    <testsuite name="Unit Tests">
        <directory>./application</directory>
    </testsuite>
    <filter>
        <whitelist>
            <file><!-- Insert path to a source file for code coverage here --></file>
        </whitelist>
    </filter>

    <logging>
        <log type="coverage-html" target="./log/report" charset="UTF-8" yui="true"
hightlight="true"
            lowupperbound="50" highlowerbound="80" />
    </logging>
</phpunit>
```

Code sample D.3.1: An example of the *phpunit.xml* file

The referenced bootstrap file is somewhat different from the systems bootstrap file. It defines the environment to be the testing environment, which allows a different database to be used than the development environment. Also, the front dispatcher will not be used as dispatching pages is done manually in tests.

The *TestCase* class is created extending *Zend_Test_PHPUnit_ControllerTestCase*. This gives access to Zend specific assertions and functions to make requests. Each other test class extends the *TestCase* class, giving it the same functionalities.

Beside general set up functionalities implemented in the *TestCase* class, it implements the *setUpBeforeClass* function, which PHPUnit automatically calls before starting test runs on a new class file. In this function, the test database is cleared and two users that are used in the tests are added to the user database, as can be seen in Code sample D.3.2. This ensures the test database is clean when starting tests in a certain test class.

```
class TestCase extends Zend_Test_PHPUnit_ControllerTestCase {

        public static function setUpBeforeClass() {
                $db = Zend_Registry::get('db');
                //clear database
                //add two users
        }

}
```

Code sample D.3.2: The *TestCase* class providing a clean environment

In a test class itself, the environment needs to be kept clean between every test function. An example of the *setUp* and *tearDown* functions of the *PaymentTest* class can be found in Code

sample D.3.3. The *setUp* function is called before every test function starts, the *tearDown* is called after each test function ends. This allows the programmer to make sure the environment is clean before each test run.

```
class PaymentTest extends TestCase {
        private $payments = null;
        private $payment = null;
        private $users = null;
        private $user = null;

        public function setUp() {
                //set up objects
                $this->payments = new Payments();
                $this->payment = $this->payments->createRow();

                $this->users = new Users();
                $this->user = $this->users->createRow();
        }

        public function tearDown() {
                //clear objects
                if ($this->payment != null)
                        $this->payment->delete();
                if ($this->user != null)
                        $this->user->delete();

                unset($this->payments);
                unset($this->payment);
                unset($this->users);
                unset($this->user);
        }
}
```

Code sample D.3.3: The *setUp* and *tearDown* classes, provided to clean up your environment

Sample code showing how Zends provided functions can be used to dispatch a page and check the content of the resulting page, can be found in Code sample D.3.4. Here the *POST* variables are set, that are used when creating a recurring payment. After dispatching the summary page, the recurring payment should be set up, with the information given in the *POST* variables. Also, the controller and action used are checked in every test.

```
public function testSummaryActionRecurringWeekly() {
        $startDate = $this->dateCurrent;
        $endDate = $this->dateTenWeeks;

        //set the POST variables for the next request
        $this->request->setMethod('POST')->setPost(array(
                                        'pay_receiver_id' => '484',
                                        'pay_amount' => '100.00',
                                        'pay_description' => 'PHPUnit test payment',
                                        'pay_recurring' => 'pay_recurring',
                                        'rpay_frequency' => PaymentFrequency::WEEKLY,
                                        'rpay_date_start' => $startDate->format('m/d/Y'),
                                        'rpay_date_end' => $endDate->format('m/d/Y')
                                ));
```

```
        //make the request
        $this->dispatch('/pay/summary');

        //assert the information is shown
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'User, Test');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '$100.00');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'PHPUnit test payment');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', 'Every week');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', $startDate->format('m-d-
Y'));
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', $endDate->format('m-d-
Y'));
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '10');
        $this->assertQueryContentContains('div#toggleDiv0 table tr td', '$100.00');

        //assert the right actions were taken
        $this->assertController('payment');
        $this->assertAction('summary');
        $this->assertModule('main');
        $this->assertResponseCode(200);
}
```

Code sample D.3.4: A test using Zend Framework specific assertions

Set up and Running tests

To run the tests, PHPUnit should be installed together with *PHP_CodeCoverage*. Various guides on how to do this are available. PHPUnit official website: http://www.phpunit.de/manual/3.0/en/installation.html

Once this is properly installed, the database needs to be set up. This database should be configured to have the name *roomplug_test*. A stripped version of the database can be found in the sql folder on Github under the name *roomplug_test.sql*. Import this file into the database and you are all set. To run the tests, go to the tests directory using your command line and enter:

```
phpunit -d memory_limit=512M
```

The extra parameter ensures the tests do not run out of memory while executing. Somehow, PHPUnit is not really efficient with memory usage.

The code coverage report can be found in HTML format in the tests/log/report folder.

Dependency on PayPal systems

One of the problems encountered with the current test suite, is its dependency on the PayPal systems. Certain tests require a connection to the PayPal systems. For example, to initiate a payment, a token is retrieved from the PayPal systems to use when redirecting to the PayPal payment flow. If the internet connection drops, or when PayPal has down time, the tests will fail. There is no real fix for this, it's up to the programmer to identify if the problem lies in the PayPal system or not.

Settings for PayPal communications

The current master branch contains links to the live endpoint of PayPal. To test, those should be changed to the sandbox endpoint. This is still the case in the *PaymentSystem* branch. Files to be checked are:
/library/paypal/Config/paypal_sdk_clientproperties.php (endpoint)
/application/modules/main/view/scripts/payment/summary.php (form action)
/application/modules/main/view/scripts/payment/summaryjs.php (redirect url)

Note: The file /application/models/Paypal.php contains some defenitions that interfere with the definitions in the config file of the PayPal SDK library. To make sure the right endpoints are used, change those too.

These files contain comments on the changes needed. Also, for identification purposes, the applicationId should be changed to 'APP-80W284485P519543T' in the following files:
/application/models/PaypalAdapter.php (search for applicationId)
/library/paypal/Config/paypal_sdk_clientproperties.php

The reason these changes have to be made is the sandbox contains two test accounts that are used in the tests while communicating with PayPal. Currently the test account used is registered by Lars. To change this, an account should be made on http://developer.paypal.com to retrieve your own API username, password and signature, which can be changed in:
/library/paypal/Config/paypal_sdk_clientproperties.php

The test accounts have the following information:
sender_1312933115_per@gmail.com
First name: Test
Last name: User
P/w: 12345678

receiv_1312932837_per@gmail.com
First name: Test
Last name: User
P/w: 12345678

Note: Specific changes done to go from sandbox to live are found in those commits:
https://github.com/roomplug/roomplug/commit/3b140412b63476deac3834c92a3eb0fae6373e30
https://github.com/roomplug/roomplug/commit/8fe7c06079c36d7aa65e2de286c784b36fe6d9c9
https://github.com/roomplug/roomplug/commit/b964d6b8bf26ee1c0e91e30026adedf6feb8d217

Appendix E: Roomplug

This appendix will give insight in the environment in which this project is executed and it will help to understand some choices that have been made during the project.

E.1 Roomplug

Roomplug is a startup company founded in 2010 and is based in San Clemente, California, USA. As landlords, both Rodney Nguyen (founder) and Sri Kanajan (co-founder) had a hard time managing their properties and their renters using existing services. They then decided to create Roomplug, a company with the mission of making online property management accessible and easy for everyone. [1]

The online rental market already had property marketplaces, background and credit check services and rent payment systems available when Roomplug was founded. Roomplug is unique because it puts all these tools and services together into one system. From finding a room or roommate, to signing the lease agreement, and paying the rent. Everything is there.

Currently the Roomplug system provides the following functionality:

1. Users can create advertisements for the rooms they have for rent, or to indicate that they are looking for a room;
2. Users can search trough the advertisements;
3. Users can apply to an advertisement, after which the owner of the advertisement can accept or decline an application or ask them for a background report;
4. Users can use the TransUnion [2] background check service to provide the landlords with a background report about themselves;
5. Users can set up a lease agreement with another user. This lease agreement can be digitally signed.

Roomplug is constantly working on improving these tools and services by getting to the users. Rodney and Sri talk to renters about their experience in both becoming and being a tenant. They also visit landlords to ask them about their experience with renting out and managing their property.

To complete the vision of having a one-stop shop for renting, Roomplug also has to work on new features. These include a property manager and an online rent payment system. This report and the project it is describing is about the payment system.

E.2 Payment System

The rent payment system is an important part of the Roomplug business since it will take away a lot of the pains of paying rent. It is common for American tenants to pay the rent by check. The landlords have to cash in all these checks at the bank each month. It would therefore save the landlords a lot of time if the tenants were able to pay the rent online. The tenants, on their turn, will save time and effort by setting up a recurring payment which automatically pays their rent each period (weekly, biweekly or monthly). They do not have to worry about paying rent in time or about sending a check to the landlord.

An online rent payment system would also help landlords and tenants in managing their rental income and expenses. The system provides a way to pay rent but also gives the landlord the possibility to reimburse the tenant for some expenses, like paying the plumber. Having an online payment system will give a clear view on the financial transactions being made between the landlord and the tenant.

E.3 Startup Company

Working in a startup environment makes things different then doing a project at a big established company. To begin with, Roomplug operates from a home office so work is only two steps away from your bed. The big differences lie in the way longer term projects are being handled.

While big corporate businesses have solid strategies worked out for the next few years, startup companies are struggling with their business plan as a whole. As Roomplug is getting closer to the market and as they talk to more and more future investors, things change around. The business model and the priority of feature requirements change constantly. This makes it harder for a software engineer to plan out a two month development track, building a payment system for instance. The used development methodology and the design choices should take these possible changes into account.

E.4 USA

The bachelors project is not only about applying software development skills. It also gives the opportunity to work abroad and experience different cultures. The preliminary and orienting part of this project took place in The Netherlands but the development of the payment system itself is completely done in San Clemente, California, USA. Getting to know the American, and especially the Californian, way of living and working is part of this project. The report will not go into this subject elaborately since it is focusing on the technical side, but it puts the project in a somewhat different context. Going abroad for this project brings some extra concerns such as harder time limitations and thus more careful planning.

References

[1] “About Roomplug”, Roomplug. Visited on July 27, 2011. <http://www.roomplug.com/about>.

[2] “Credit Report, Credit Score, Credit Check – TransUnion.com”, TransUnion. Visited on Sept 30, 2011. <http://www.transunion.com>.

Appendix F: Software Improvement Group feedback

F.1 Initial feedback

De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De score wordt naar beneden gehaald door de Unit Interfacing en de Unit Size.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Voorbeelden van methodes met een bovengemiddeld aantal parameters in dit systeem zijn bijvoorbeeld de methodes 'recurringInputHasErrors' en 'createRecurringPayments' in 'PaymentController'. Vanuit de beschrijving van de parameters valt af te leiden dat alle parameters samen een 'RecurringPayment' vormen. Om de interface van deze methode duidelijker te maken zou hier wellicht een object van het type RecurringPayment aan meegegeven kunnen worden. Verder zijn er meerdere methoden die een 'startDate' en een 'endDate' meekrijgen. Het grouperen van deze parameters in een 'Period'-object zou ervoor kunnen zorgen dat het duidelijker wordt wat de relatie tussen deze twee variabelen is.

Voor Unit Size wordt er gekeken naar het percentage code dat zich bevind in methodes welke bovengemiddeld lang zijn. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor te onderhouden wordt. In dit systeem valt op dat er enkele templates zijn waarin PHP-code, HTML-code en Javascript code door elkaar heen is geschreven. Het is aan te raden om te kijken of dit van elkaar gescheiden kan worden om het zo voor toekomstige ontwikkelaars duidelijk te maken waar welke logica is geïmplementeerd.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden zodra het systeem verder groeit. De aanwezigheid van test code lijkt er in ieder geval op te duiden dat het aanpassen van de huidige code zonder al te veel moeite kan plaatsvinden.

F.2 Feedback on improvements

In de tweede upload zien we dat de omvang van het systeem is toegenomen, maar dat de score voor onderhoudbaarheid licht is gedaald. Uit de metingen blijkt dat de score voor Unit Interfacing flink is gestegen, de meegestuurde documentatie laat ook zien dat hier goed mee omgegaan is. Wat betreft de templates zien we dat de Javascript uit de templates is verwijderd, maar dat er nu meer langere (en deels gedupliceerde) templates zijn. Hierdoor zijn de scores voor Unit Size, Unit Complexiteit en Duplicatie gedaald. Omdat deze dalingen het positieve effect van de Unit Interfacing opheft komt het totaal uit op een lichte daling van de score voor onderhoudbaarheid.

Uit deze observaties en de meegeleverde documentatie kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject. Daarnaast scoort het systeem absoluut gezien bovengemiddeld en laat de meegeleverde documentatie zien dat de studenten serieus om zijn gegaan met de aanbevelingen.

Appendix G: Digital documents

This appendix includes all documents that are too large to be delivered on paper (in size or in numbers). The documents can be found on the DVD attached to this report.

G.1 Source code

The source code of all code written during this project is can be found in the 'Appendix G1' folder on the DVD.

G.2 Extended design documents

Most of the design diagrams can be found in chapter 4. Some of them were originally more extended then the diagrams that are shown. The extended versions were too big to paste in the report itself. That is why simple versions were created and the extended versions are added here as appendices. They are added digitally because they will not fit on one page and splitting them up defeats the purpose of having one diagram.

These diagrams can be found in the 'Appendix G2' folder on the DVD:

- Model class diagram - Milestone 1
- Model class diagram - Milestone 2
- Model class diagram - Milestone 3
- Controller class diagram - Milestone 1
- Controller class diagram - Milestone 2
- Controller class diagram - Milestone 3
- PayPal payment page flow - extended

G.3 phpDocuments

All code produced is extensively documented. A guide is generated describing the classes and functions based on this documentation. To generate this guide, phpDocumentor is used, a tool to generate those kind of reports [1]. The phpDocuments can be found in the 'Appendix G3' folder on the CD.

References

[1] "phpDocumentor: The complete documentation solution for PHP", phpDocumentor. Visited on October 8, 2011. <http://www.phpdoc.org/>.